AUTOMATED EXTRACTION OF TRANSITION SYSTEMS FROM COMPONENT-MODEL ARCHITECTURES

by

MATTHEW W. HOOSIER
B.S., Kansas State University, 2002

A REPORT
submitted in partial fulfillment of the requirements for the degree
MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Engineering
KANSAS STATE UNIVERSITY
Manhattan, Kansas
2004

Approved By

Major Professor
Matthew B. Dwyer
ABSTRACT

Modern software design heavily utilizes concurrency to divide work into logical streams of execution, improve performance, and distribute an application across multiple processing nodes. A frustrating consequence of this rise in concurrent processing is the difficulty in reasoning about multithreaded programs’ behavior. The chief irritant to analysis is a rapid fan-out of the possible computational states in which a multithreaded program may find itself. An increasingly popular formal method to grapple with this combinatorial explosion of program states is an automatic state-space explorer, or “model checker.”

Model checkers have gained wide use in hardware design, network protocol analysis, and other domains. Conspicuously, little work has been done to model-check distributed middleware-based software. This report presents a translator for use with the Cadena component-model analysis tool, which automatically produces structural models of the dataflow and middleware in a Cadena system configuration. We demonstrate that automatically compiled models for the Bogor model checker can utilize domain-specific extensions to the core model-checking algorithms to efficiently explore the state space of distributed object systems.
# TABLE OF CONTENTS

List of Figures iii  
List of Tables v  
Acknowledgements vi  

1 Model Checkers 3
   1.1 Motivations ................................................. 3  
   1.2 Finite State Systems ....................................... 4  
   1.3 Exploring the Reachable State Space ....................... 5  
   1.4 The Bogor Model Checking Framework ....................... 6  
   1.5 Customizing Bogor to a Domain ........................... 8  

2 Component Architectures 10
   2.1 Introduction ................................................. 10  
   2.2 The CORBA Component Model ............................... 12  
      2.2.1 Structural Primitives ................................. 13  
      2.2.2 Messages Types ........................................ 14  
      2.2.3 Ports .................................................... 14  
      2.2.4 Modes .................................................... 15  

3 Cadena Systems 17
   3.1 Real-time Embedded Avionics Systems ....................... 17  
   3.2 Structural Description ..................................... 18
## LIST OF FIGURES

1.1 Classical stateful depth-first search used in model checkers ........................................ 6  
1.2 Dining philosophers in the BIR guarded transition language ........................................ 7  
1.3 Language extensions for a BIR set ............................................................................. 9  
   
2.1 Simple CCM system type definitions ........................................................................ 13  
2.2 Simple CCM system instantiated ............................................................................. 13  
2.3 CORBA component with a mode variable .................................................................. 15  
   
3.1 Structural design artifacts for ModalSP (excerpts). In IDL ........................................ 19  
3.2 ModalSP system instantiated .................................................................................. 20  
3.3 ModalSP behavioral description (excerpts) ............................................................. 21  
3.4 ModalSP assembly description in CADL (excepts) .................................................... 23  
   
4.1 BIR extension definition for publish/subscribe infrastructure ..................................... 28  
4.2 BIR extension definition for client and server interface ports .................................... 30  
4.3 BIR functions to invoke receptacle methods ............................................................. 31  
4.4 Example event correlators in Cadena ........................................................................ 32  
4.5 Hypothetical direct BIR implementation of correlators .............................................. 33  
4.6 BIR language extensions for event correlators ......................................................... 33  
4.7 BIR API for manipulating CPS variables .................................................................... 35  
4.8 Excerpts from system initialization in BIR ................................................................. 42  
4.9 BIR threads for generating timeouts .......................................................................... 45  
4.10 Scheduling a Prism system to use rate monotonicity and prune frame overruns .......... 46
5.1 Comparison of resource requirements
# LIST OF TABLES

5.1 Test case scenario configurations ........................................... 47
5.2 Experimental data ................................................................. 49
ACKNOWLEDGEMENTS

I would like to thank several people for their support throughout my university education. First, my parents, Dale Hoosier and Cindy Villarreal. They have always offered unwavering confidence and encouragement in my planning and actions, and never complained too loudly about my infrequent visits home. Next, my academic adviser, Dr. Dan Andresen. For the past four years, he has been a willing soundboard for my questions and patiently offered career advice. Finally, my friends. I appreciate the feedback that we have provided each other on projects both academic and personal. Drawing on the wisdom of others who have been through the same trials makes the experience much more bearable.
Introduction

Contemporary software is increasingly built in a distributed paradigm. What once was a monolithic application is now factored into several reusable (it is hoped) pieces. The state-of-the-practice incarnation of this trend is the component-model architecture. The component architecture framework, whose most popular version this report introduces later in Section 2.2 encapsulates the application logic in a self-contained unit termed the component.

Despite a long-standing preference for relying on simple, lightweight technologies, some makers of safety-critical systems are beginning to embrace the use of off-the-shelf middleware systems. The aircraft industry, for example, has undertaken an initiative to standardize its embedded avionics control systems on popular and well understood frameworks such as the Corba Component Model [8]. These embedded control platforms used are typically multithreaded and use some real-time variant of a standard middleware to meet quality-of-service requirements.

While the switch to standard distributed-object systems does much to reduce the amount of tailoring that must be done to migrate existing code to new architectures or hardware configurations, some of the most challenging aspects left of these distributed real-time event-driven (DRE) systems are their concurrent behaviors. Verifying that complicated requirements which relate aspects of system behavior affected by multiple threads can be deceptively hard using traditional quality-assurance techniques such as test-case construction and simulation. The determinism inherent in concurrent programs makes constructing sufficient test cases for all thread interleavings impractical.

A technique known as model checking [4] developed as a way to analyze communications hardware protocols (which share much of the same susceptibility to subtle errors as concurrent programs) avoids the methodology of test cases entirely. Rather, an abstract description of a system’s concurrent behavior is fed into the model checker. This specialized program systematically generates every possible scheduling decision which may occur at runtime. Able to assess every possible threading
schedule, the model checker then determines if the model meets specifications. This provides a very strong level of assurance that concurrent systems are correct with respect to particular formulae identified by designers.

In response to a dearth of design tools to assist developers of component architecture software, researchers at Kansas State University have built a tool named Cadena to allow high-level system configuration, analysis of real-time execution properties, and even code generation for several popular middleware environments [11]. One novel feature of Cadena is its use of model checking to verify aspects of intended DRE program behavior. Although model checking has been seldom used for distributed system analysis because of the state-space explosion problem which is described in Chapter 1, Cadena uses an extensible model checker, Bogor [18], which allows customization to the application domain (in this case, event-driven real-time software) to increase abstraction and avoid checking scheduling decisions when can be proven a priori not to occur at runtime. Using these two features, Bogor has been used to model check Cadena designs efficiently [7].

While this use of model checking in a previously uncharted domain is promising, the input programs for the Bogor model checker were laboriously constructed manually by model-checking experts. As a proof of concept, this is fine. It does not, however, enable product-line engineers to employ model checking as a standard technique. The task of creating the somewhat cryptic inputs to a model checker must be automated for this quality assurance mechanism to gain acceptance. This report describes the features and implementation of a compiler\(^1\) to translate Cadena scenarios to BIR (the input language to Bogor) programs for fully automatic use in the production release of Cadena.

Specifically, this report (i) describes the constructs and domain customizations of Bogor used to encode the behavior of DRE systems as input to the model checker, (ii) presents comparisons of the performance of the existing hand-optimized models to their automatically compiled counterparts, and (iii) argues that the overhead introduced by automatic generation of BIR programs is acceptable, and that the technique scales well as system complexity increases.

\(^1\)This tool is integrated into Cadena, available at \texttt{http://cadena.projects.cis.ksu.edu}.
Chapter 1

Model Checkers

Put simply, a model checker is a specialized software verification tool which accepts as input a program and a logical formula which prescribes a desired behavior for the program, and determines whether the program is a “model” of the formula. This checking is entirely automatic; the user must intervene only to create the input program to the checker and to write the logical specification.

1.1 Motivations

Clarke et al. paint a disheartening assessment of regular software quality assurance techniques in their book [3]. The world, we are told, is growing ever more dependent on computing to provide the necessities of life. Commerce depends on mission-critical transaction servers, aviation relies on fault-intolerant control software to safely chart courses and monitor aircraft progress, and the rockets that are our space vehicle launch platforms (upon whose payloads commerce and science increasingly lean) plot flight vectors without room for errors.

Traditionally, software engineers have employed simulation and test cases to ensure software quality. While effective, these arts become increasingly difficult as software systems scale in complexity and turn to concurrency (more than one simultaneous stream of execution). Concurrent programs—also called multithreaded programs—are especially challenging to test because one is never guaranteed which stream, or thread, of control in the program will execute on the processor at a given point. As the number of possible interleavings of instructions from multiple threads grows, test case designers face a complex system that quickly exceeds standard human capacity for understanding.
One immediately wonders then if some automated means of program verification exits. Unfortunately, this effort is foiled in its infancy by the nemesis of many computer science efforts: the halting problem [13]. This import theorem asserts that no program can exist which determines if an arbitrary program \( p \) (possibly itself!) terminates on all possible inputs. Since termination would be a basic requirement of correctness for most programs to be verified, a completely automated verifier is not possible.

By restricting the scope of our ambition, though, one can prove behavioral characteristics of an important subset of programs: those that have finite state. This is a somewhat limiting restriction (e.g., no arbitrarily large data structure may exist), but nonetheless most safety-critical systems (or simple abstractions of them) fit into this class [3]. Next we provide some terminology and definitions for speaking about model checkers.

### 1.2 Finite State Systems

In this section, we appeal to a mathematical description of a computer system as a transition system. Unless otherwise noted, all definitions are drawn from [5].

The state of a program consists of the values of all its variables. In the realm of transition systems, the term “variable” is understood to include both explicit variables (e.g, \( x \) and \( y \)) and implicit variables (thread program counters). A description of the state of a program forms a complete snapshot of the system from which it may indistinguishably be restarted later.

A transition describes the flow of a program from one state to the next. In concrete terms, this may occur by way of a variable assignment (\( x \leftarrow 0 \)), an explicit program counter change (\( \text{goto } 10 \)), or even a null operation (even here, the program counter changes).

A finite state transition system, then, is a 4-tuple \( (S, T, S_0, L) \). \( S \) is the set of all possible states for the system, \( S_0 \) is the set of initial states (usually a singleton), and \( L : S \rightarrow 2^{AP} \) is a relation mapping a state to the set of all primitive atomic propositions which hold on it. \( T \subseteq S \times S \) is the set of all transitions between states in the system. We will often refer to transitions enabled when a program is in state \( s \); this simply means the set of all transitions defined from the current state: \( \{(s_1, s_2) \in T \mid s = s_1\} \).

Any program state in the transition system other than a state start must be reached by a nonempty sequence of transitions. We call this ordered sequence of transitions with intervening
states a path, and denote it as $\pi = s_0 \xrightarrow{t_1} s_1 \ldots \xrightarrow{t_n} s_n$.

Now a transition $\alpha$ in the enabled set of a state $s$ can be from any thread running in the system. In all but the most trivial examples, many states have transitions from multiple threads enabled. It is precisely this fact that several threads are able to proceed at any one time which causes the combinatorial blowup of program states that justified our inquiry into model checking in the first place. In order to successfully decide if the prescribed specification holds on any possible execution of the system, we must first have a method for reliably visiting all the reachable states.

### 1.3 Exploring the Reachable State Space

A model checker operates on a transition system. Visiting all the reachable states (those states which are accessible by a feasible sequence of transitions beginning at the start state) of the model is in reality a deceptively simple task. A depth-first search (DFS) is initiated on the start state, with the enabled transitions serving as arcs between nodes. Because multiple distinct paths from the start state $s_0$ may lead to the same state $s$, the DFS must be stateful.

To cope, the DFS maintains a set of seen states as it descends through the tree of possible execution paths. Before each new subtree is traversed, the DFS first examines the seen state set to determine if this successor state has already been visited. If it has, then the search is pruned because we are already guaranteed that every successor of the pruned node will have been visited earlier when the pruned node was first explored. Figure 1.1 contains a listing of the simple procedure to visit all reachable states in a transition system.

This algorithm can easily be proven to visit all the reachable state space of a concurrent system, and trivial changes allow it to enforce invariants and detect deadlock.

More complicated variations on the simple DFS allow a model checker to check that transition systems satisfy sophisticated temporal logic formulae [12]. These temporal logics usually are sufficiently expressive to allow quite complicated requirements to be expressed; in fact, the problem facing specifiers is usually that of how best to encode the desired specifications into the temporal logics. Dwyer et al. have done some work on developing a database of commonly used specification patterns [9]. If these authors’ projections are correct, this may enable normal developers (i.e., people who are not specification experts) to write reasonable logical specifications for programs, which will then be verified with model checkers.
1.4 The Bogor Model Checking Framework

Many model checkers have been implemented, each with its own technical strengths. SPIN—perhaps the best known of them—is widely used in industry and academia. SPIN implements many of the most advanced optimization algorithms known to handle the state explosion problem [12]. SPIN’s input language Promela does support a fairly expressive set of datatypes, including native handling of integral types. Unfortunately, SPIN has no facilities for dynamic memory allocation or arbitrary pointers. This makes modeling general-purpose software cumbersome when possible at all.

SMV and its successor NUSMV [2] achieve great scalability to systems with enormous state spaces (over $10^{100}$ states, far beyond the general-case ability of SPIN). This efficiency comes at a cost, though: the input language to SMV does not even support regular integers. This forces engineers to perform extensive abstraction of regular software (which, oddly enough, often uses ints) before one may even consider analyzing its properties with SMV and its relatives.

Both SPIN and SMV are extremely valuable for analyzing low-level systems with urgent safety requirements, such as hardware systems and communications protocols. Hardware designs, operating mostly on the level of boolean values, are ideally handled in SMV. Communications protocols, with their finite set of entities and limited use of integral datatypes, have been analyzed for almost two decades by SPIN and its predecessors.

To verify application-style software though, direct support for the object oriented programming features employed would be useful. To address this need, the Bogor model checking framework [18]
Figure 1.2: Dining philosophers in the BIR guarded transition language

was developed at Kansas State University.

Bogor is a modularized model checker which processes an input language, BIR, which is amenable to direct support for object oriented concepts such as rich datatypes, dynamic creation for both objects and threads, virtual method dispatch, and space reclamation by automatic garbage collection. BIR is a guarded command language; this means that each transition is optionally preceded by a boolean-valued condition which must hold for the transition to be enabled. Figure 1.2 shows a simple BIR model of the dining philosophers problem.

The dining philosophers system illustrates several principles of the BIR language. The reader will notice the use of record structures as complex data types. Though not pictured here, one record type may extend another and implicitly inherit all the parent record’s fields; this provides a mechanism for object-oriented type hierarchies.

BIR has two facilities for atomic actions (atomic blocks in the Promela language). First, all
actions executed inside a single do block are atomic by definition, since a loc is a single thread control point that corresponds to a discrete program counter value. For example, the array to contain the forks is allocated, and each fork is initialized inside the transition leading out of control point loc0. There is no intermediate state between the beginning and ending of this initialization. Second, if one needs to span multiple do blocks inside an atomic step, then the invisible keyword may be used. We use this feature to force the spawning of all Philosopher threads to be an atomic action. There is some subtlety at work here. Although the MAIN thread’s program counter does advance as the thread progresses from loc0 to loc1 up through loc3, no intermediate states are saved, and the scheduler continues executing MAIN until the end of the invisible transition sequence is reached. Thus, no intermediate results are visible to any other thread.

1.5 Customizing Bogor to a Domain

Bogor’s real novelty lies not in its direct support for object-oriented features—other model checkers such as dSPIN [14] first added these features. Rather, it is the easy customization of Bogor to application domains that makes analyzing standard programs more tractable.

One may introduce new extension types into a BIR model by a special forward declaration which both provides a name and a series of operators for manipulating the new type. This allows abstract datatypes needed for models (but whose internal state is irrelevant to the correctness properties of the system) to be treated as a “black box”, dramatically reducing their overhead in the model checker’s state vector. Additionally, all manipulations of extension types are performed in an atomic step. This prevents an unnecessary state-space blowup from intermediate forms of the data structure. Figure 1.3 shows the definition of a new extension set type for BIR.

In addition to the data-hiding benefits of extension types, Bogor is implemented with a modular internal design. Robby et al. remark that this is key differentiating factor over, e.g., SPIN, because it allows one to develop domain-specific search strategies. [18]. This is particularly helpful, for example, in the analysis of embedded systems (which often use some highly predictable scheduling policy such as rate monotonic). During early work using Bogor, Deng et al. were able to reduce the state space of embedded systems model checks by multiple orders of magnitude by writing a custom thread-scheduling module for Bogor that ignored impossible thread schedules [7] which the vanilla scheduler would nonetheless have explored.
system SetExample {

   extension Set for setpackage.SetModule {
     
      // this is a parameterized type; sets may hold any
      // type of element
      typedef type<’a’>;

      // side effect-free expressions
      expdef Set.type<’a’> create<’a’>();
      expdef ’a choose<’a’>(Set.type<’a’>);
      expdef boolean isEmpty<’a’>(Set.type<’a’>);
      expdef boolean contains<’a’>(Set.type<’a’>);

      // state-transforming actions
      actiondef add<’a’>(Set.type<’a’>, ’a’);
      actiondef remove<’a’>(Set.type<’a’>, ’a’);
   }

   Set.type<int> intSet;

   main thread MAIN() {
      loc loc0: live {} 
      do {
         intSet := Set.create<int>();
      } goto loc1;

      loc loc1: live {}
      do {
         Set.add<int>(intSet, 0);
         Set.add<int>(intSet, 1);
         Set.add<int>(intSet, 2);
      } goto loc2;

      loc loc2: live {}
      do {
         // check for implementation bug!
         assert (Set.contains<int>(intSet, 1));
      } return;
   }

   
}

Figure 1.3: Language extensions for a BIR set [18]
Chapter 2

Component Architectures

2.1 Introduction

One may observe that an ever-increasing proportion of new software projects are distributed applications. Rather than executing from start to finish in a traditional address space on one host, these applications delegate portions of the work they must accomplish to an outside program—which is itself considered part of the overall “program.”

The classic example of such a system is a simple client-server arrangement. In this paradigm, the main logic is encapsulated in a long-running “server” process. One requests this program to perform some work by invoking a “client” program which formulates some request. This request is then communicated to the server by one of many inter-process communication (IPC) techniques. Typically, this is done with a TCP/IP network connection. After acknowledging this request, the server performs the appropriate business logic to form an answer and then reports the results of the computation back to the client through the IPC mechanism used earlier.

This simplest form of a distributed application has perhaps the best-known example in the world among its practitioners: the World Wide Web infrastructure. All webpages are requested over a simple client-server system known as the HyperText Transfer Protocol (HTTP). In this scenario, the browser (e.g., Internet Explorer or Mozilla Navigator) formulates a request which consists of the name of a page the user wishes to view. This request is transferred out of the browser’s address space by a TCP/IP network connection to the long-running server (the “web server”), which then goes about the necessary tasks to prepare the webpage.
It is worth noting at this point that even this most basic example exhibits some abstraction between the client and server. The browser does not care by what means the web server constructs the contents of the requested page. There are several immediate options:

- The page may be read directly from an unchanging disk file;
- The page may be extracted in its entirety from a database;
- A template may be instantiated with certain data values, to generate a customized result page;
- or
- Other more complicated procedures involving heavy processing and transformation may be invoked to create the page.

We will return to this issue of abstraction later.

These most simple client-server protocols rely on data transfer conventions that are completely ad hoc for each new application. The developer is entirely responsible for developing a scheme to encode (marshal) data into some network-amenable form and symmetrically a scheme to decode (unmarshal) the network-formatted data into something meaningful to the remote process. This becomes a burden and source of errors to developers contemplating a complex, new distributed application.

An early framework for alleviating this barrier to entry came in the form of Remote Procedure Calls (RPC) [6]. Here, the developer is encouraged to view the act of using IPC as a simple function call, as in high-level languages. After specifying a contract (standardized list of data items exchanged during the IPC), one invokes a code generation tool that encapsulates the complexity of network data encoding and decoding behind a pair of simple programming-language functions (one each for the client and server). These functions are then the developer’s interface to the IPC. Their usage is (or aims to be) indistinguishable from regular API calls in the program.

RPC represented a great advancement over hand-rolled communications techniques. It is the most rudimentary form of what has come to be known as middleware: a stand-alone infrastructure which manages the low-level details of communication between programs. From this simple beginning, a frighteningly large collection of middleware systems has emerged: SOAP, RMI, XML-RPC, JMS, EJB, DCOM, and CORBA are but a small sampling of the familiar names in this collection.
Most of these middleware frameworks converged on an object-oriented view; rather than contextless simple function calls (e.g., `foo()`), the client instead makes a method call on a remote object with internal state (e.g., `obj.bar()`).

One distributed object middleware in particular has achieved widespread acceptance in industry: the CORBA (Common Object Request Broker Architecture) framework. In addition to the expected features for a middleware system (i.e., shielding the developer from low-level networking and transport concerns), CORBA provides at least two key features.

First, the details necessary for implementing a CORBA system (i.e., the specification) are an open standard. By making these details public, one ensures that engineers using CORBA are not tied to a particular vendor’s toolchain—they are free to switch compilers and runtime environments as necessary.

Second, CORBA is language-independent. Once one has written the external interface for a CORBA object in the Interface Definition Language (IDL), the ORB (CORBA runtime environment) vendor’s tools generate source code bindings for the system in whatever language desired. In practice, the language independence of IDL is often exploited as a means of allowing data import/export from programs (previously incompatible) which have been retrofitted with CORBA support.

Recent trends have seen middlewares move toward an even higher layer of abstraction: the component model architecture. In this arrangement, the central unit of computation and deployment is a component, which is analogous to an object with well-defined points of entry and egress from its boundary (ports). By restricting all communication with the component to these specific, few locations, it is thought that a better level of modularity and deployment flexibility is achieved.

### 2.2 The CORBA Component Model

After a late start, the CORBA specification was revised to include a component architecture layer. The resulting CORBA Component Model (CCM) is a superset of the expressiveness of standard CORBA. The chief additions are the focus on a component as the computational unit and an event service for sending messages between components [17].

The example IDL system in Figure 2.1 and the system configuration instantiated over it in Figure 2.2 will motivate our discussion of the primary features in CCM. This toy system reflects a three-component configuration where one component, the “consumer”, receives notification from
eventtype NewDataAvailable {};

interface DataSource {
    readonly attribute string data;
};

interface Worker {
    void doWork(string datapoint);
};

component Producer {
    provides DataSource dataOut;
    publishes NewDataAvailable outDataChanged;
};

component Actor {
    provides Worker action;
};

component Consumer {
    uses DataSource dataIn;
    uses Worker action;
    consumes NewDataAvailable inDataChanged;
};

Figure 2.1: Simple CCM system type definitions

a data “producer” that a new data point is available. The consumer then directs a “worker” to perform some processing on the new data point.

2.2.1 Structural Primitives

CCM, like its predecessor CORBA, features “interfaces” as the aggregation point for operations on an object. Our simple example defined two interfaces: DataSource and Worker. Note that in our example, the DataSource interface features an attribute named data. This is only a notational convenience; attributes in IDL are expanded by the code generator into pairs of accessor and mutator

Figure 2.2: Simple CCM system instantiated
methods (e.g., `get_data()` and `set_data()`). The presence of the `readonly` keyword is a compiler directive that specifies only the accessor `get_data()` should be generated.

The `component` construct in CCM IDL is used to collect a set of named interface instances into a single logical unit of computation. For instance, our `Consumer` component requires a source of input (`dataIn`) and requires an object to “do something” about the input (`action`).

The other two components, in contrast, provide objects for other components to use. The `Producer` owns a source of data (`dataOut`) to which other components may connect and utilize. The `Actor` component owns an object (`action`) which itself knows how to handle some unit of input.

### 2.2.2 Messages Types

CCM components communicate by exchanging one of two message types: interface messages and event messages. The more familiar of the two is the interface message. It is a synchronous, unbuffered communication whose content is an integral part of the logical program flow. All inter-component method calls are communicated via interface messages. In our example, a call to the `doWork()` method on `Worker` is implemented using a synchronous interface method; this provides the intuitive semantics that the call has completed by the time that control flow returns to the calling component’s code.

Alternately, a component may use an event message, which is an asynchronous, optionally buffered communication. Events are used in a CCM system to serve notification that some application-specific condition of interest has occurred. In our example, the producer informs the consumer that the current datapoint has changed by issuing a `NewDataAvailable` message to all interested parties. In contrast to the synchronous behavior of interface messages, note that the issuer of an event is made no guarantee that all (or any) receivers have processed a notification by the time that control flow returns to the component which outputs an event.

### 2.2.3 Ports

We have—till now—been speaking informally about the roles of objects declared inside components by referring to them as requiring or providing some functionality from or to the outside world.

Formally, CCM encodes this dependency information in its notion of ports, or points of external communication on a component. Ports are partitioned into two sets: event ports and interface ports.
Event ports are the external locations on a component through which it emits or receives event
notifications (the asynchronous messages). Interface ports are the points at which a implementation
of an IDL interface—for example, DataSource— is exposed to the outside world. By defining an
interface port, a component either announces to the world that it provides a particular service (e.g.,
dataOut on Producer) or that it requires the use of another component’s service (e.g., dataIn on
Consumer).

The communication which a component conducts with the outside world is limited to (1) sending
or receiving messages over event ports and (2) making calls against the “provided” interface on the
port of another component. This allows the CCM developer to completely express the functional
requirements of a component in high-level terms and permits great implementation flexibility.

The connections among ports are unidirectional; that is, only the “client” role is permitted to
initiate communication. These connections are additionally strongly typed; the declared type of
the “server” end on a port-to-port link must be assignment-compatible with the declared type
of the “client” end. For example, the outDataChanged port on Producer has a static type of
NewDataAvailable. The connection between Producer.outDataChanged and Consumer.inDataChanged
depicted in Figure 2.2 is legal because outDataChanged’s type is equal to inDataChanged’s type.

2.2.4 Modes

Although not pictured in Figure 2.1, it is legal and often desirable to declare simple attributes directly
on a component’s top-level definition. Consider the definition of SimpleComponent in Figure 2.3:

```
enum ActivityFlag { enabled, disabled };
component SimpleComponent {
    attribute ActivityFlag status;
};
```

Figure 2.3: CORBA component with a mode variable

In this case, status is a variable which can be directly queried at the top level of the component;
that is, it is not contained within any port on the component, and no navigation is required to access
it.

As we shall explore in more detail later, engineers in certain application domains often turn to
using simple attributes in the style of status (above) to enumerate which of several runtime states
a component occupies. The component’s internal behavior (which is strictly not described at all in CCM IDL) varies in predictable ways depending on the value of all such state-indicating attributes which live on the component [7]. For now, we simply note the existence of this design pattern and refer to such attribute variables as a component’s *mode variables*. 
Chapter 3

Cadena Systems

The Cadena [11] tool, developed at Kansas State University, aims to enable engineers to perform high-level analysis on component-model software systems. The present version of Cadena uses the Corba Component Model (CCM) as a standardized mechanism for describing system architectures. This structural model is then supplemented with lightweight specifications which describe the dataflow dependencies between components.

With this system description in hand, the engineer may then ask Cadena to perform various structural analyses:

- Determine the set of components affected by downstream propagation of events.
- Query for the mode combinations during which event propagation may be squelched entirely.
- Automatically assign each component to a rate group based on the arrival frequency of its incoming events.
- Perform an explicit-state model check of a transition system representative of the system configuration, to determine whether various safety and liveliness requirements are satisfied.

3.1 Real-time Embedded Avionics Systems

It is a somewhat surprising trend, embedded systems engineers are beginning to embrace middleware solutions to build multiprocessor control systems. Hatcliff et al. remark on this:

\footnote{A proprietary architecture description language is planned in the near future.}
Even in the domain of distributed real-time embedded (DRE) systems where hard/soft deadlines and minimal footprint requirements traditionally have led developers to eschew sophisticated middleware solutions, component-based infrastructures are growing more popular because hardware advances allow real-time and embedded requirements to be more easily achieved. In addition, component-based infrastructures provide a framework for systematically introducing important domain aspects such as time-triggered notification, real-time scheduling, and fault tolerance. [11]

Despite the growing acceptance of communication middleware in control-systems middleware, though, engineers have relatively few tools to automate their designs. For instance, Rational Rose (the predominant industrial software modeling tool) lacks support for CCM.

Boeing’s Prism component system is a case where embedded systems meet a component-based design methodology. Interestingly, the Prism metamodel, while closely related to CCM, is not directly compatible with CCM: some generalization features of the full CORBA Component Model are omitted as inapplicable in an embedded application, and other domain-specific features above and beyond CCM are added as optimizations to enhance schedulability proofs and runtime performance. Furthermore, Prism’s pre-CCM roots give rise to some structural incompatibilities.

As a consequence of these differences, a standard CCM modeling tool (if such a program existed) would not directly satisfy Boeing engineers’ analytical needs. Cadena’s creators cite this as a core motivation for tailoring the tool to Boeing Prism applications. The remainder of this report will discuss Prism systems explicitly as a representative sample of embedded real-time component-based systems.

### 3.2 Structural Description

Despite the occasional structural incompatibility between CCM and Prism, it suffices to use the standard IDL from the CORBA Component Model as an architecture-description language in Cadena. The incompatibilities take two forms:

1. Architectures allowed in CCM but not Prism. These are easily handled by rejecting such configurations with the structural type-checker built into Cadena.
Figure 3.1: Structural design artifacts for ModalSP (excerpts). In IDL.

2. Extra domain-specific features in Prism. These features are either (a) additional deployment information, or (b) extra configuration of the communication links between components. Cadena addresses these by introducing a separate design artifact called a profile which allows systems engineers to force component developers to attach extra specifications of these required features to system configurations. This effectively augments the architecture language to reach the degree of richness needed for Boeing domain-specific purposes.

Figure 3.1 shows some representative excerpts from a small Boeing Open Experimentation Platform (OEP) configuration referred to as ModalSP. The model is given its name because it utilizes several “modal” components (refer to Section 2.2.4) and runs on a single processor (SP).

Note the heavy use of an event infrastructure to notify components when data values on which they depend have been updated (inDataAvailable, outDataAvailable). This reflects a design pattern known as control-push data-pull [11], in which components only update their data values immediately after receiving notice that a new value is available. This confers the advantage that the data-consuming components never need to block their threads during data acquisition, since they have just prior received notification that data is already available. If one is not convinced this is a valuable property a priori, a discussion later will offer reasons why the thread executing a component’s code should not become blocked in mid-method.
3.3 Behavioral Description

IDL files define the structure of a system. This, however, is hardly enough information to extract a transition system suitable for model checking (the eventual subject of this report) or perform port-triggering dependency analysis. Cadena introduces an auxiliary specification format, the Component Property Specification (CPS) to supplement the structural information encoded in IDL.

A CPS file structurally follows the module definitions as given in IDL. Rather than declaring type information, though, it presents finite state machine-like dataflow descriptions inside each method on a component. An example of the CPS syntax for the primary modal component of ModalSP is given in Figure 3.3.

We can see that the behavioral description of the BMLazyActive component gives insight into its runtime mode switches. Suppose C is a BMLazyActive component. Then C can be in one of two macro-states: (a) a current copy of some input data is cached, or (b) the cached copy of said input data is stale. The component tracks this with its mode variable bmLazyMode, which is bound to the IDL variable BMLazyActive.dataStatus (see Section 2.2.4). If bmLazyMode indicates that C has a current copy of the data, then it serves out the cached value whenever consumers request a datapoint reading via the accessor on dataOut.data (a facet method). If—on the other hand—a
module common {
  component BMLazyActive {
    mode bmLazyMode represents dataStatus init common.LazyActiveMode.stale;
    behavior {
      string buf;
      accessor string dataOut.data() {
        case bmLazyMode of {
          common.LazyActiveMode.stale:
            // refresh the data, reset mode
            buf := dataIn.data;
            bmLazyMode := common.LazyActiveMode.fresh;
          common.LazyActiveMode.fresh:
            // do nothing extra
        }
        return buf;
      }
      handle inDataAvailable(DataAvailable e) {
        // reset mode to "stale"
        bmLazyMode := common.LazyActiveMode.stale;
        // tell data consumers that new value is available
        push new DataAvailable {} on outDataAvailable;
      }
    }
  }
}

Figure 3.3: ModalSP behavioral description (excerpts)

notification that new data is available has occurred since the last refresh of the cache, then C must
first reacquire its data by consulting its receptacle attribute-accessor method dataIn.data().

The syntax of a CPS specification gives rise to some interesting observations. First, what appears
to be an assignment (buf := dataIn.data) is in fact a dataflow trace. The grammar of CPS is
purposely left not rich enough to encode any computations on data values like buf. To model such
transformations would be inappropriate for high-level analysis and (as a matter of practicality) would
likely quickly cause a state-space explosion in a model check of the underlying transition system.
Thus we satisfy ourselves with observing that a statement such as buf := dataIn.data will record
the fact that a data value update occurred, but will not take notice of the particular values passed
through such calls.

Second, the set of operations available to the CPS writer (most are demonstrated even in this
small example) form a core of observable actions which will eventually form the atomic propositions
in a temporal property specification which should hold over a Cadena system. The indivisible
observable actions which we anticipate as useful are items such as “method call begun,” “method
call returned,” “port connection traversed,” “data value assigned,” “mode value changed,” “event
broadcasted,” and similar events. This work is ongoing and not the direct subject of this report.

3.4 System Assembly

A vanilla CCM system infrastructure provides no human-readable language for describing the instantiations of components and interconnections among their ports. Rather, the whole of system deployment and configuration is described by the Component Assembly Package: essentially, a large zipfile containing several XML files which are read and automatically processed by a CCM runtime environment [16]. The OMG’s intent is for individual CCM system vendors to provide proprietary tools which generate the Component Assembly Package as an output artifact. Conversations with embedded avionics component developers convinced Cadena developers that a textual system assembly language would be enthusiastically received.

Recall the previous brief discussion about “value-added” system configuration options that Prism engineers required in order to carry out optimizations and schedulability analyses. These data items generally include the following flavors of information:

- The embedded processor on which a component will be hosted.
- Tags to indicate whether a component is a master or proxy. Proxy components—which mirror the behavior of a master component—are often placed on the same embedded processor board as consumers of the data if the master component is hosted on a remote board and would therefore make frequent communication with many consumers extremely expensive.
- Annotations indicating the synchronization disciplines which should be observed when consumers access or modify a component’s data.
- Which priority of thread should service (dispatch) pending event-handler code for particular event sink ports.

This information is all domain-specific and beyond the capabilities (and intentions) of the CORBA Component Model to retain or handle appropriately.

These two independent rationales led the Cadena developers to design a tool-specific system configuration language: the Cadena Architecture Description Language (CADL). A novel feature of CADL is its ability to require domain-specific annotations (such as those listed above regarding
synchronization, replication, and threading) without hard-coding these into the language grammar. Figure 3.4 shows an example of the syntax of CADL.

3.5 Middleware Environment

As may be expected of a specialized embedded system, the execution environment for Prism is quite different than the standard multiprogrammed, general-purpose operating system for enterprise component-based applications (e.g., Enterprise Java Beans). Rather, such avionics systems run in a real-time event channel with some peculiar behaviors.

The event channel performs much of the administrative work of broadcasting events and invoking the appropriate handler methods of receivers. Event-producing ports do not maintain their own subscriber lists; this is handled internally in the event channel. Rather, it (the port) only maintains a reference to a proxy port in the event channel. This allows components to be devoid of boilerplate code for multiplexing and demultiplexing event propagation.

All computation in the system is initiated by regularly scheduled timeout events; each scheduled timeout event typically has a frequency between 1 Hz and 40 Hz. Further, the timeout rates in a system are harmonic; that is, if $r_1$ and $r_2$ are the Hz values of two timeout frequencies in a system then $(r_1 > r_2) \rightarrow (r_1 \mod r_2 = 0)$. This guarantees that the rate of any one timeout event is evenly divisible by the rate of any lower-frequency event.
The components in a Prism system are passive; there is no thread dedicated to the execution of any one component’s code. Rather, the system employs a pool of threads. For each distinct timeout rate in the system, one distinguished thread exists in the pool to dispatch the handler methods of components’ ports which subscribe to the timeout. For a timeout occurring at a frequency of \( r \) Hz, we refer to the thread responsible for dispatching associated events as the “run-rate \( r \)” thread.

One interesting consequence of the passive threading model is the requirement that every event connection in a Prism system must be tagged as belonging to a particular run-rate. This follows from the requirement that fact that every thread in the thread pool is assigned to a particular run-rate; hence, one must statically designate which thread is to service the event-handling ports on each component. Boeing engineers report that it significantly eases the prediction of runtime behavior when components are not contending for a nondeterministic assignment of event-handling threads.

The static assignment of run-rate threads to every event-handling function gives rise to the need for buffering of events inside the real-time event channel, for it frequently occurs that a high-priority thread (hence, a thread which will not relinquish the processor) wishes to generate an event headed for a consumer port serviced by a low-priority thread.

This case is handled by maintaining a queue of pending event handler methods for each thread to dispatch. In the scenario we hypothesize, the high-priority thread simply enqueues the handler function of the recipient into the run-rate queue for the lower-priority thread and continues on until its work in the frame is finished or it is involuntarily preempted. Low-priority threads may enqueue work for higher-priority run-rates in an analogous way.

In contrast to the complicated semantics of event delivery, interface calls from a receptacle to a facet are quite simple. Because the control-push data-pull pattern is used, such data-retrieving interface calls are guaranteed not to block. An event-dispatch threads simply invokes the appropriate method on the remote component’s facet and performs its computation directly.
Chapter 4

Modeling Approach

One driving factor behind the development of Cadena was the desire to provide automated model checking of various lightweight, high-level properties that Boeing engineers find difficult to reason about using traditional distributed real-time embedded (DRE) development tools. For example, it is quite hard to design test cases which exercise simple properties encoded in the following English specifications:

(a) “At all times, exactly one of the tactical steering and navigational steering components will be in an enabled mode.”

(b) “When a 20 Hz system timeout occurs and the navigation steering mode is selected, the display component will be updated by data from the navigation component rather than the tactical component.”

Preliminary work has been done to check some example Prism systems as modeled in Cadena with respect to these sorts of properties. In [11], Promela models were constructed for the dSPIN object-oriented customization of SPIN ([14]) and automated exploration of the state space of Prism systems was first achieved. Here, the investigators were able to verify simple properties such as (a) and (b) above. The difficulty here was that dSPIN lacked sufficient domain customization to prevent a frightful explosion of RAM requirements even for a small system like ModalSP (pictured in Figure 3.2).

A subsequent round of model-checker analysis on Prism systems used the Bogor [18] model checker (see Section 1.4 for a brief discussion of its novel features). By implementing DRE domain-
specific language extensions to BIR, Cadena developers achieved dramatic scalability results over the dSPIN effort [7].

While successful at proving (and disproving) invariants over Prism systems, the collection of models from this analysis was constructed by hand. With the average model for anything but the most unrealistically small Prism system requiring at least 2,500 lines of BIR markup, this quickly becomes a daunting task for researchers. It is completely impractical for product-line engineers. Moreover, even if the engineers managed to construct completely faithful representations in BIR of their system assembly scenarios (doubtful), it is questionable whether the price-performance ratio is favorable for them to do so. Clearly some automation is warranted.

This report describes the implementation of a fully automated Cadena to BIR translator which can—at the click of a button—generate BIR models and explore them with Bogor. Much of the reusable portion of the BIR models (primary the DRE event channel with its publish/subscribe mechanisms) is directly reused or inspired by the methods used in [7]. We nonetheless describe them for completeness and to document any design drift which has occurred since their original forms.

4.1 Overview

Modeling a Cadena DRE system which features a loosely coupled coupled component architecture can be broken into four main tasks:

1. Encoding the high-level port interconnections while retaining the modularity of components.

2. Capturing the behavioral characteristics of the components' methods (e.g., event handlers and interface methods). This means transliterating the CPS dataflow notation into the modeling language.

3. Dealing with the flow of time and the boundaries of successive computation frames without blowing up the state space.

4. Initializing the system; that is, allocating component instances and interconnecting them before the timing requirements take effect.

The first of these goals is handled by implementing a reusable publish/subscribe infrastructure as a collection of BIR language primitives. Task 2 is accomplished by means of a recursive-descent
compiler. The third issue is—in the author’s opinion—the most challenging; we discuss later a scheduling policy we implemented for Bogor which leverages some key schedulability assumptions to avoid exploring infeasible interleavings. Last, we address task 4 above by performing a (possibly extensive) system initialization as the first action in the models’ main thread; only after this the static structure is established are other threads spawned.

4.2 Event Channel Infrastructure

4.2.1 Publish/Subscribe Event Mechanism

Like CCM, components in a Prism system route their communication through event ports and interface ports on their exterior. The translator implementation lifts these concepts to first-class language primitives with the extensions listed in Figure 4.1. We have introduced a Component primitive which internally maintains a subscriber list for each of its event-pushing ports. As seen in the listing, several operations are available to manipulate subscriber lists. addSubscriberList() hooks a new outgoing port onto the exterior of a component; individual subscribers may then register with this new event source by using addSubscriber().

A library function, fireEventFromComponent() (only its signature is shown) is responsible for multiplexing an event emitted from the port of a component. While the code for this event-multiplexing function is long, its task is simple: for each subscriber to the source port given by getSubscribers(), either (a) if the subscriber is in the same rate group (see Section 3.5 for a discussion of the threading model) then directly dispatch the event by calling the subscriber’s event-handler method, or (b) if the subscriber is in a different rate group, then store the event into the pending events queue for the thread which dispatches the subscriber’s run rate.

Individual subscribers to events consist of Subscriber records; each of these encapsulates the reference to a subscriber component and a function which should be called as the method handler.

Notice also that we introduce an extension type to encapsulate the actual event itself: Event. An instance of this type (created by the createEvent() constructor expression defined just below) is passed along from the point where the source component emits an event all the way to the handler method on the subscriber. At present, this event type only encapsulates the runtime type of the event (recall that CCM systems have a separate type hierarchy for event types). By adding a language primitive for the event value type passed to handler functions, though, this provides an
extension CAD for edu.ksu.cis.projects.cadena.bogor.ext.SystemModule {

    // two new nonprimitive value types
    typedef Event;
    typedef Component;

    // component constructor
    expdef CAD.Component createComponent(
        string // component's name
    );

    // event constructor
    expdef CAD.Event createEvent<a>(
        'a // an enumerated value encoding the event type
    );

    // create an event-publishing port on a component
    actiondef addSubscriberList(
        CAD.Component, // component to host the event source
        string // name of event source port
    );

    // register a subscriber to an event-source port
    actiondef addSubscriber<a>(
        CAD.Component, // component on which events source lives
        string, // name of event source port
        'a // the new subscriber
    );

    // retrieve all subscribers to an event-source port
    expdef 'a[] getSubscribers<a>(
        CAD.Component, // component on which the port lives
        string // port's name
    );

} // extension CAD

record Subscriber {
    EventHandlerType handlerFunction;
    string portname;
}

record ComponentSubscriber extends Subscriber {
    CAD.Component component;
    boolean isSynchronous;
    int dispatchRate;
}

function fireEventFromComponent(
    CAD.Component sourceComp,
    string port,
    CAD.Event event)
{
    ...
}

Figure 4.1: BIR extension definition for publish/subscribe infrastructure
easy migration path for future application which pass data values as payload of the event.

4.2.2 Interface Ports

Providing an implementation of the event propagation infrastructure only solves half of task 1 listed above; a mechanism to route calls against receptacle methods as they are encountered in CPS, to the particular BIR object and function which provide the receptacle implementation is still needed. The machinery used to implement the synchronous ports is less complex than its event-multiplexing cousins: essentially, each receptacle port on a component stores an object reference to the facet port to which it is connected. As shown in Figure 4.2, the BIR API for creating the facet ports and hooking them to receptacles consists of only a few calls. This infrastructure is completely reusable from one Prism model to the next.

The BIR code to invoke a facet method call, however, is not completely generic. Because there must be a separate virtual function table for each different interface method (e.g., FooInterface.bar() and FooInterface.baz() will require separate virtual function tables), a unique BIR function for each interface method must be stamped out from a template. In Figure 4.3 we see two possible instantiations of the function which handles routing interface calls from client to server. Fortunately, customizing the handoff function for each distinct interface method is an easy task; we use a Velocity [1] parametric template and simply specialize it with appropriate type names as needed.

4.2.3 Event Correlation

An important aspect of event propagation has thus far been entirely ignored: event correlation. In publish-subscribe frameworks, consumers often wish to know when something more complicated than a singleton event has been generated. A sensor-monitoring component, for example, may wish to know when all $n$ of its data-collection units has generated new data. The practice of filtering for such higher-level logical formulae over events is called event correlation. In [19], a convincing case is made that DRE systems already make heavy use of event correlation, with projections showing a steadily growing trend.

In order to support logical event correlation in the Prism initiative, Cadena employs a correlator system tailored for the CORBA Component Model. Unlike some other event correlation frameworks, the Cadena correlators directly accept and produce CCM-compatible messages. This makes the
extension CAD for edu.ksu.cis.projects.cadena.bogor.ext.SystemModule {

    // two new nonprimitive value types
    typedef Event;
    typedef Port;

    // component constructor
    expdef CAD.Component createComponent(
        string // component's name
    );

    // facet (server) port constructor
    expdef CAD.Port createPort();

    // hook a facet port onto the hosting component
    actiondef registerPort{
        CAD.Component, // hosting component
        CAD.Port, // the facet port
        string // name of the port
    };

    // retrieve a facet port by name
    expdef CAD.Port getPort(CAD.Component, string);

    // register the BIR function that provides the implementation of
    // some method on a facet port
    actiondef setPortMethodHandler<"a">(
        CAD.Port, // the facet port
        string, // name of the method (e.g., "foo")
        "a" // value in virtualtable's enumerated type
    );

    // find the enumerated type value that points to a BIR function
    // implementing a method on a facet port
    expdef getPortMethodHandler<"a">(
        CAD.Port, // facet port
        string // name of the method (e.g., "foo")
    );

    // links a client (receptacle) port to a server (facet) port
    actiondef connectPorts{
        Pair.type< // client port's host and name
            CAD.Component,
            string >,
        Pair.type< // server port's host and name
            CAD.Component,
            string >
    };

    // look up the component and name of a port on it, to which
    // a client port is hooked up
    expdef Pair.type<CAD.Component, string> getProvider{
        Pair.type<
            CAD.Component,
            string >
    };
}

Figure 4.2: BIR extension definition for client and server interface ports
A string-returning interface method

```java
function { F o o I n t e r f a c e . foo () < invoke > ||| } ( C A D . C o m p o n e n t consumer ,
  string portName ,
  string methodName)
returns string
{
  Pair . type<
    CAD . Component ,
    string > remotePair ;
  CAD . Component receiver ;
  string remotePortName ;
  CAD . Port remotePort ;
  { { F o o I n t e r f a c e . foo () } } handlerFunction ;
  string result ;
  loc loc0 : live {} // list omitted
do {
    remotePair :=
      CAD . getProvider (  
        Pair . create<
          CAD . Component ,
          string > ( consumer ,
            portName ));
  } goto loc1 :
loc loc1 : live {} // list omitted
when remotePair != null do {
  provider := Pair . first<
    CAD . Component ,
    string > ( remotePair );
  remotePortName := Pair . second<
    CAD . Component ,
    string > ( remotePair );
  remotePort := CAD . getPort ( provider ,
    remotePortName );
  handlerFunction :=
    CAD . getPortMethodHandler<
      { { F o o I n t e r f a c e . foo () } } ( 
        remotePort ,
        methodName ));
  } goto loc2 :
when remotePair == null do {
  // no connected facet
  return result ;
} goto loc2 : live {} // list omitted
result := invoke virtual
  { { F o o I n t e r f a c e . foo () } } ( 
    handlerFunction ,
    provider )
return result ;
}
```

A void-typed interface method

```java
function { Bar I n t e r f a c e . bar () < invoke > ||| } ( C A D . C o m p o n e n t consumer ,
  string portName ,
  string methodName)
{
  Pair . type<
    CAD . Component ,
    string > remotePair ;
  CAD . Component receiver ;
  string remotePortName ;
  CAD . Port remotePort ;
  { { Bar I n t e r f a c e . bar () } } handlerFunction ;
loc loc0 : live {} // list omitted
do {
  remotePair :=
    CAD . getProvider (  
      Pair . create<
        CAD . Component ,
        string > ( consumer ,
          portName ));
  } goto loc1 :
loc loc1 : live {} // list omitted
when remotePair != null do {
  provider := Pair . first<
    CAD . Component ,
    string > ( remotePair );
  remotePortName := Pair . second<
    CAD . Component ,
    string > ( remotePair );
  remotePort := CAD . getPort ( provider ,
    remotePortName );
  handlerFunction :=
    CAD . getPortMethodHandler<
      { { B a r I n t e r f a c e . bar () } } ( 
        remotePort ,
        methodName ));
  } goto loc2 :
when remotePair == null do {
  // no connected facet
  return ;
} goto loc2 : live {} // list omitted
invoke virtual
  { { B a r I n t e r f a c e . bar () } } ( 
    handlerFunction ,
    provider )
return ;
```

Figure 4.3: BIR functions to invoke receptacle methods
Simple filter expression
\[
\text{DataAvailable } \text{correlation} \text{ SimpleAnd}( \\
\text{DataAvailable e1,} \\
\text{DataAvailable e2}) \\
\text{e1 + e2} \\
\{ \\
\text{case true:} \\
\text{push new DataAvailable {}}; \\
\}
\]

Complex filter expression
\[
\text{DataAvailable} \\
\text{correlation} \text{ ComplicatedFilter(} \\
\text{DataAvailable e1,} \\
\text{DataAvailable e2,} \\
\text{DataAvailable e3}) \\
\text{(e1 + e2) | (e2 ; e2) | e3} \\
\{ \\
\text{case true:} \\
\text{push new DataAvailable {}}; \\
\}
\]

Figure 4.4: Example event correlators in Cadena

insertion of such a filter into the middleware completely natural and fits within a unified type system. The full semantics and syntax of the Cadena event correlation framework is given in [15].

It will suffice to mention here that the event filters in Cadena take on a regular expression-like syntax (it is shown formally in the presentation that they are regular). This includes the capability for arbitrary nesting of logical subformulae. For example, both of the correlators given in Figure 4.4. In practice, most RT event channels implement an event correlator—regardless of the framework—as a variation of an automaton.

The choice of modeling strategies for these event correlators proved something of an enigma: the most intuitive representation in Cadena’s scheme is to allocate a BIR variable which records whether each atomic input event has been satisfied. One imagines then stamping out a higher-level nested BIR expression to determine whether the entire filter is satisfied. It might look something like the proposed implementation in Figure 4.5.

Such a direct implementation using BIR native variables, though, would become extraordinarily messy when confronted with the general case of the correlator language, which includes a transformer phase (not discussed here) that performs post-processing before generating output events. Even a correct generation of BIR code to implement this would unnecessarily bloat the state space (because multiple Bogor transitions would be needed to process a single event) and the state vector (all the machinery variables to implement the correlator would become part of the bit-vector).

BIR language extensions provide a much more elegant mechanism to model correlators. The API to the language extension is shown in Figure 4.6. We have used a novel method for configuring the filter and transformer portion of the correlator without over-complicating the API; a complete
boolean e1_received;
boolean e2_received;
...

function handle_e1_received(Correlator.type correlator)
{
    loc loc0: live {} do {
        e1_received := true;
    } goto loc1;
    loc loc1: live {}
    // filter not satisfied
    when !(e1_received && e2_received) do {} return;
    // filter satisfied
    when e1_received && e2_received do {} goto loc3;
    loc loc3: live {}
    invoke fireEventFromCorrelator(correlator)
    goto loc4;
    loc loc4: live {} do {
        e1_received := false;
        e2_received := false;
    } return;
}

Figure 4.5: Hypothetical direct BIR implementation of correlators

extension Correlator for edu.ksu.cis.projects.cadena.bogor.ext.CorrelatorModule {
    // new value type
typedef type;

    // constructor
expdef Correlator.type create(
        string // full specification language
    );

    // process an incoming event
actiondef handleEvent<'evtType>(
    Correlator.type, // the correlator
    string, // name of input event (e.g., "e1")
    List.type<'evtType> // place where any events generated should be placed
    );

    actiondef addSubscriber<'a>(
        Correlator.type,
        'a
    );

expdef 'a[] getSubscribers<'a>(
    Correlator.type
    );
}

record CorrelatorSubscriber extends Subscriber {
    Correlator.type correlator;
}

Figure 4.6: BIR language extensions for event correlators
string giving the entire semantic specification of the correlator (as seen in Figure 4.4) is passed as an argument to the Correlator.create() constructor. This may seem a heavy-handed and unforgiving solution, but it works quite well. There is no difficulty in obtaining the complete semantic specification of the correlator; this is a compile-time artifact already available to the Cadena-to-BIR translator. Processing the specification text in the Java code which implements the constructor is likewise not an onerous task; we simply reuse the correlator parser already bundled with Cadena. While parsing a correlator at model-checking time seems an expensive operation, it is only done during one transition (during system initialization, on the transition out of the initial state) and does therefore not confer any noticeable runtime penalty on Bogor.

As the state-space exploration of a Prism system with correlator proceeds, the Correlator.type value primitive has its internal state transformed by successive calls to the handleEvent() operator. Both internal state resets and emission of outgoing events to the downstream subscribers are handled automatically (the latter by populating the List.type instance passed to the event-handling operator).

In order to integrate with the infrastructure we discussed earlier (which uses instances of the Subscriber record as the unit of event subscription), we define a subtype of Subscriber, called CorrelatorSubscriber, which stores a reference to the appropriate correlator. This enables seamless dispatch of events to subscribers, whether they are correlators or components.

### 4.3 Method Behavior Transliteration

Having introduced our modeling technique for the structural aspects of Cadena systems, we next turn our attention to soundly encoding the dataflow behavior of components. In Cadena, the structural IDL component definitions are supplemented with the Component Property Specification (CPS), which gives the modal and dataflow properties of each method per component.

Our general strategy for handling this is to treat the sequence of CPS statements inside a method as a list. At the topmost level, we specify that the tail of the list (and implicitly all its predecessors) should be compiled to BIR; every CPS instruction preceding the list tail is handled recursively.
4.3.1 Data Storage

Before presenting the translation strategies for each of the specific CPS constructs, we first remark on the runtime structures used to store each category of value which may be used in the expressions of a CPS statement.

CPS variables are updated and accessed by a dedicated set of operators in the BIR language extension for components. This set of polymorphic operators (pictured in Figure 4.7) handles both mode variables (e.g., bmLazyMode) and private buffer variables (e.g., buf) as seen in Figure 3.3. For example,

```plaintext
loc loc0: live {}
when CAD.getAttribute<string>({||this||}, "buf") == "foo" do {} return;
```

Method call results are modeled faithfully as the results of calling real BIR functions with non-void return types. Note that the values IDL attributes are computed in this way (in addition to explicit IDL operations) because the IDL language bindings will compile l-value and r-value uses of attributes into accessor and mutator method calls. A peculiarly of Bogor’s handling of program counters forces us to compute a function’s value and do subsequent testing in two distinct control points:

```plaintext
loc loc0: live { x }
   x := invoke {||common.ReadData.data|get|<invoke>()|}({||this||}, "dataIn", "data<get>")
goto loc1;
loc loc1: live { x }
   when x == "foo" do {} return;
```
Formal arguments to an IDL method or event handler are modeled using the actual formal arguments to the BIR function implementing the event handler or facet method. The simple identifier of the BIR formal argument is used to retrieve the formal argument’s value.

```java
function {{common.FakeComponent.doSomething(int)}}{
  {{common.FakeInterface.doSomething(int)}} {{enumValue}},
  CAD.Component {{this}},
  int arg1)
{
...
  loc loc5: live {} 
      when arg1 > 10 do {} return;
      when !(arg1 > 10) do {} goto loc6;
  ...
}
```

Literals such as the enumerated values are cast as BIR enumerated values. The stale value of `bmLazyMode` in our CPS snippet, for example, is modeled as the `{{stale}}` element of the `{{common.LazyActiveMode}}` BIR enumerated type. For example, to test if the value of the `bmLazyMode` mode variable is stale, we have the following:

```java
loc loc0: live {} 
    when CAD.getAttribute<{{common.LazyActiveMode}}>({this}, "bmLazyMode") == {{common.LazyActiveMode}}.{{stale}} do {
    return;
}
```

This collection of CPS-to-BIR identifier mappings—when combined with the usual boolean and precedence operators—is sufficiently expressive to compile any CPS expression into equivalent BIR code.

### 4.3.2 CPS Intra-method Behavior

At the heart of a CPS’s dependency specification is its description of the flow of data values inside a component. A dataflow statement captures this movement of values: it may describe the acquisition of data from remote sources via accessor method calls on receptacle ports, forcibly transporting the data to another component by mutators on receptacle ports, or internal copying (via variables like buf from the ModalSP example) or broadcasting to all subscribers that new data is available.
To this set of data-tracing primitives, CPS adds some control flow constructs: a familiar if statement and a case switch that operates on mode variables. The combination of describing data movement and modal behavior allows Cadena to capture high-level behavioral transitions that Prism engineers wish to analyze.

Each of these CPS operators is readily expressed in BIR with the help of our language extensions seen previously.

If Statements

As expected, “if” statements in CPS do the usual thing; a boolean-typed expression is evaluated, and control flow branches depending on its truth value. Formally, CPS if constructs have the form

\[
\text{if ( } E \text{ ) then } \{ B_1 \} \text{ else } \{ B_2 \}.
\]

These are compiled into BIR fragments using the following scheme:

\[
\begin{align*}
\ldots \\
\text{boolean } e_{\text{val}}; \\
\ldots \\
\text{loc loc0: live } \{ e_{\text{val}} \} \\
\quad \text{do} \{ \\
\quad \quad e_{\text{val}} := \text{valueOf}(E); \quad // \text{last stage of recursive evaluation} \\
\quad \} \text{ goto loc1;} \\
\text{loc loc1: live } \{ e_{\text{val}} \} \\
\quad \text{when } e_{\text{val}} \text{ do } \{ \} \text{ goto locB1;} \\
\quad \text{when } ! e_{\text{val}} \text{ do } \{ \} \text{ goto locB2;} \\
\text{loc locB1: live } \{ \} \\
\quad \ldots \text{ goto locNext;} \\
\text{loc locB2: live } \{ \} \\
\quad \ldots \text{ goto locNext;} \\
\text{loc locNext: live } \{ \} \\
\quad // \text{control flow continues after the if statement}
\end{align*}
\]
Case Statements

These constructs are a special branching mechanism that operates over tuples of mode variables. Suppose we have the following case statement:

```plaintext
case (m_1, m_2) of {
  (v_1^a, v_2^a): B_1;
  (v_1^b, v_2^b): B_2;
}
```

Then this is realized as BIR in the following pattern:

```plaintext
boolean b1_match; boolean b2_match;
...
loc locCompileExp: live { b1_match, b2_match }
  do {
    b1_match := valueOf(m_1 == v_1^a) && valueOf(m_2 == v_2^a);
  } goto locCompileExp2;
loc locCompileExp2: live { b1_match, b2_match }
  do {
    b2_match := valueOf(m_1 == v_1^b) && valueOf(m_2 == v_2^b);
  } goto locTest;
loc locTest: live { b1_match, b2_match }
  when b1_match do {} goto locB1;
  when b2_match do {} goto locB2;
  when !(b1_match) && !(b2_match) goto locNext;
loc locB1: live {}
...
goto locNext;
loc locB2: live {}
...
goto locNext;
loc locNext: live {}
  // computation resumes
```
Concrete Assignment

A concrete assignment assigns a particular value deterministically to some l-value compatible variable. It has the form `<id> := <exp>`. We do not list all the possible permutations of l-values (port attributes, mode variables) and r-values (method call results, literals, buffered variables, mode variables). The construction of the assignment is a simple combination of the previous techniques shown for evaluating expressions and updating the values of buffer- and mode-variables. Evaluating the r-value and assigning to the l-value are completely orthogonal.

Abstract assignment

This is a novel feature of CPS that allows capturing of the dataflow between numerically oriented devices (e.g., sensors) and the components that perform business logic based on this data. In syntax, it appears similar to a concrete assignment. For example, `int x := dataIn.getSomeString()`.

Notice, however, that the “assignment” need not be type-safe. This is because no actual values are copied during an abstract assignment. Rather, either a dummy default value (for r-values whose domain is unbounded) or all possible values (for r-values whose domain is an enumerated type) are nondeterministically assigned to the l-value.

Cadena designers often use abstract assignment to record the fact that data values are retrieved by a component, which then does extensive post-processing. Thus, the abstract assignment avoids the requirement of modeling the data transformations. While this may at first give the reader pause about the fidelity of representing component’s behavior in CPS, Prism engineers have verified that sensory data values do not affect the modal behavior of components. Thus, we soundly capture the behavior relevant to model checking.

For example, if the mode variable `myMode` of the type `enum OnOffType { on, off }` is abstractly assigned a string value which is returned by a port method call `dataIn.getData()`, the following BIR would be generated:

```c
string temp_val;
...
// fetch the r-value, so that all the port dependencies
// are still preserved
loc loc0: live { temp_val }
    temp_val := invoke {} |DataInterface.getData()<invoke>|()
```
// now nondeterministically give the l-value every possible
// value of the enumerated type
loc loc1: live {}
do {
    CAD.setAttribute<OnOffType>({||this||}, "myMode", OnOffType.on);
} goto loc2;
do {
    CAD.setAttribute<OnOffType>({||this||}, "myMode", OnOffType.off);
} goto loc2;

// computation continues
loc loc2: live {}
...

Event Propagation

Event-pushing statements—as their name implies—are the CPS construct used to signify that a
cOMPonent emits an event message from one of its event source (“publishes”) ports. The BIR code
to accomplish this is concise; it simply chains together calls to the library functions and language ex-
tensions from the re-usable middleware representation. For example, if the current component emits
a DataAvailable event from its outDataAvailable port, this is realized by the following BIR fragment:

loc loc0: live {
    invoke fireEventFromComponent(
        {||this||},
        "outDataAvailable",
        CAD.createEvent<{{Event|}}>({||Event|}, {{common.DataAvailable|}}))
    return;
}

The constructs we have seen—logical flow control, assignment, and event pushing—are the en-
tirely of CPS’s expressiveness. All CPS method behavior definitions can be compiled to BIR transi-
tion systems by composing the translation templates defined here. We have now seen the methods
by which messages are propagated to subscribers and interface calls are routed to server ports, and
the techniques for expressing the contents of the handler and methods which handle these messages.
Only one final stage of the Cadena-to-BIR translator remains to be explained: system and assembly initialization.

4.4 System Initialization

Most component-architecture systems—CCM included [17]—use a standard system initialization phase. In this stage, the middleware services are brought online, component are allocated, and connections (both asynchronous publish-subscribe and synchronous interfaces) are made among the components.

In the runtime of a DRE application, it is the ORB’s API which is invoked to configure and initialize the distributed application. Since we have introduced a reusable set of BIR language extensions to mimic the relevant middleware functionality, is is the middleware-simulation Bogor API which we use analogously during the initialization phase of a Prism transition system. Components are constructed, ports are allocated and hooked to the components, and interconnections are established between ports. Additionally, we set each mode variable to its default value.

Figure 4.8 shows highlights of the system initialization for our familiar ModalSP scenario. Only a small fraction of the actions necessary to configure even a trivial system have been shown; one can gain a feeling for the large amount of configuration done at system start-up. Despite such complication, this is not a significant expense in the model checking. All configuration happens during an atomic step when only the main thread is active: the initial system thread. Only once the system is completely configured are the other threads for the pool and system clock spawned.

4.5 Time

The most challenging aspect of modeling a Prism system is to accurately represent the flow of time. In the real-time event channel, timeouts are generated by the system at regular rates (e.g., 1 Hz, 5 Hz, and 10 Hz). These timeouts are the root of a cascade of messages (in the form of event publications and synchronous method calls) that form jobs that each rategroup-handling thread must complete before the next timeout (i.e., the arrival of the next frame). This regular structure and prioritization of event-dispatch threads map well onto a rate monotonic scheduling discipline; this is the scheduling policy used at runtime for Prism systems.
// make GPS component and its ports
GPS := CAD.createComponent("GPS");
{ | tempPort | } := CAD.createComponent();
CAD.registerPort(GPS, { | tempPort | }, "dataOut");
CAD.setPortMethodHandler({ | common.ReadData.data<get> | },
   "data<get>",
   { | common.ReadData.data<get> | },
   { | common.BMDevice.dataOut.data<get> | });
CAD.addSubscriberList(GPS, "outDataAvailable");

// make GPS subscribe to 20 Hz timeout
{ | tempComSub | } := new ComponentSubscriber;
{ | tempComSub | }.handlerFunction :=
   EventHandlerType.({ | common.BMDevice.timeOut<handler> | });
{ | tempComSub | }.portName := "timeOut";
{ | tempComSub | }.component := GPS;
{ | tempComSub | }.isSynchronous := false;
{ | tempComSub | }.dispatchRate := 20;
CAD.addSubscriber<Subscriber>(EventChannel, "timeOut20", { | tempComSub | });

// connect AirFrame "dataIn" receptacle to GPS "dataOut" facet
CAD.connectPorts(
   Pair.createComponent<
      CAD.Component, string>(AirFrame, "dataIn"),
   Pair.createComponent<
      CAD.Component, string>(GPS, "dataOut"));

// make a correlator (argument is one big long run-on string!)
{ | tempCor | } := Correlator.createComponent("common.DataAvailable correlation BinaryAnd(
   common.DataAvailable e1,
   common.DataAvailable e2) e1+(32)
   { case true:
      push new common.DataAvailable {}
   });

// set default modes
CAD.setAttribute<
   { | common.LazyActiveMode | }>(
   AirFrame,
   "bmLazyMode",
   { | common.LazyActiveMode | }, { | stale | });

// start system event timeout issuers ...
start clock();
start timeoutSender();

... and threadpool
start rateDispatcher(5);

Figure 4.8: Excerpts from system initialization in BIR
If one were to attempt to model the entire behavior of the scheduling mechanism and verify the schedulability of a scenario configuration, this would require the model checker to have a worst-case bound on the time required for every action. This is problematic, for several reasons. First, model checkers are very bad at explicitly representing time. This explodes the state space with extra data and is generally antagonistic to the goal of abstracting the system. Second, CPS actions do not even attempt to capture the processing activities of a component. Engineers would be forced to annotate models with execution time estimates, and the attraction of using CPS as a simple logical description of control flow would be lost. Finally, systematically exploring the feasibility of every possible moment for a new frame to arrive (timeout) would force the exploration of every possible thread interleaving. While this complete exploration is necessary for the general case of model checking with no knowledge of the actual scheduling policies, we can do significantly better in such a tightly controlled environment as a Prism runtime.

Fortunately, Boeing engineers have very good tools already available to reason about the schedulability of their systems. A combination of high-level analysis tools tailored for DRE systems and plain testing allow the developers to independently satisfy themselves that the system meets timing requirements. The allows us to make a liberating assumption in Cadena-to-BIR models: we perform our modeling as though no frame overruns occur. This allows a clearer focus on the prescribed task (modal properties) and allows a dramatic reduction in the number of thread interleavings. The original presentation of hand-crafted Prism model checking in [7] gives a detailed treatment of the possible strategies for reducing the system branching. This report will not repeat it here, but rather outline the scheduling techniques taken from it which the automated model translator uses.

**Controlling relative timeout arrival frequency** allows us to maintain the harmonic ratios of frame lengths as seen in the actual Prism systems. If each beginning-of-frame event were allowed to arrive completely nondeterministically, then the resulting model would have traces which, for example, featured 10 times as many 1 Hz events as 5 Hz events. These traces are clearly infeasible, and stripping them out allows a sound removal of impossible interleavings.

**Strict rate-monotonic scheduling of event dispatchers** rejects on face any interleaving in which two event-dispatch threads are runnable and the model checker chooses to execute the low-priority thread. Absent this optimization, one would expect a state-space blowup roughly exponential in the number of rategroups active for a scenario.
Presumed absence of frame overruns allows the event dispatchers to run to completion before new timeouts arrive. So long as schedulability is independently verified, this optimization removes the obligation to explore traces where more than one timeout is pending for any one rategroup.

4.5.1 System Timeouts

The first construction to introduce in the model of timeouts is the actual mechanism of their injection into the event-handling infrastructure of the system. One can imagine several options for doing this: directly enqueuing the events into each runrate’s pending-events message buffer, introducing a new primitive for raw timeouts, and reusing the existing message infrastructure all come to mind.

The translation uses the existing message-publishing API introduced earlier in Section 4.2.1. Specifically, we introduce a pseudocomponent named Timer from which system timeouts appear to emanate. This nicely encapsulates the set of jobs which are initiated by the start-of-frame event (in implementation, a Prism system’s periodic tasks are executed by simply calling the push method of the component which subscribes to the timeout) as a simple series of subscribers to an event-source port such as timeOut5 or timeOut40 on Timer. The thread responsible for generating the timeouts simply calls the fireEventFromComponent() BIR library procedure on the Timer component to initiate periodic processing. Small instantiations of the two threads used to control timeout production are shown in Figure 4.9. Note the fact that we have separated the abstract notion of clock progress (a “tick” of globalClock) and the logic for emitting timeouts when the clock hits predefined values. This modularity will be leveraged to eliminate interleavings corresponding to frame overruns (which we have by assumption disallowed).

4.5.2 Scheduler Modification

The BIR construction shown allows us to explicitly control the relative frequency of the various system timeouts. A 5 Hz event, for example, is only generated when the system clock is a multiple of $\frac{\text{hyperperiod}}{5}$. Similarly, a 20 Hz timeout would be issued only at times when the system clock is a multiple of $\frac{\text{hyperperiod}}{20}$. Because we have implemented our clock, globalClock, as a wrap-around integer whose value is always in the range $[0, \text{hyperperiod} - 1]$ this guarantees that timeouts will be issued in the correct proportions.
thread clock () {
    loc loc0: live {} do { globalClock := (globalClock + 1) % hyperperiod } goto loc0;
}

thread timeoutSender () {
    int localTime;
    loc initLoc: live { localTime } do { localTime := -1; } goto stallLoc;
    // only one round of timeouts per clock tick, please
    loc stallLoc: live { localTime } when localTime != globalClock do invisible {} goto locInvokeTest1;
    // fire 1 Hz event if clock is multiple of complete hyperperiod
    loc locInvokeTest1: live { localTime } when globalClock % (hyperperiod / 1) == 0 do invisible {} goto locInvoke1;
    when globalClock % (hyperperiod / 1) != 0 do invisible {} goto locInvokeTest2;
    loc locInvoke1: live { localTime } invisible invoke fireEventFromComponent ( Timer, "timeOut1",
        CAD.createEvent <{| EventType |}> ({| EventType |}.{| common.TimeOut |}))
        goto locInvokeTest2;
    // fire 5 Hz event if clock is multiple of hyperperiod / 5
    loc locInvokeTest2: live { localTime }

    ... loc resetLocalTime: live { localTime } do { localTime := globalClock; } goto stallLoc;
}

Figure 4.9: BIR threads for generating timeouts

Some of the scheduling optimizations described above cannot be achieved by BIR code alone, however. Strict monotonic scheduling of the event dispatching threads, for example, requires selectively disabling all transitions on a thread dynamically. For this, the Cadena-to-BIR translator bundles a custom scheduler module\(^1\). This is easily done by inspecting the runtime information for each thread, determining which are event dispatchers (and at what priorities) and then simply deleting all enabled transitions for rategroups except those belonging to the highest priority such thread.

Additionally, this scheduling module is used to prune any branches on the state graph which correspond to a frame overrun. We accomplish this by partitioning the threads in a Prism system into three classes: the timeout issuer, event dispatchers (one per timeout frequency), and the global clock maintainer. The intuition behind the scheduling algorithm (shown in Figure 4.10) rests on the observation that the timeout-sending thread (timeoutSender) can only cycle once for every time that the clock ticks. By controlling exactly when the clock is allowed to tick, then, we control the conditions during which the system timeouts are generated. To prevent frame overruns, we simply

\(^1\)To be precise, the custom scheduler is an implementation of Bogor’s ISchedulingStrategist interface.
\begin{verbatim}
if timeoutSender has enabled transitions
    schedule timeoutSender thread
else if an event-dispatch thread rateDispatcher has enabled transitions
    schedule highest-priority event dispatch thread
else
    schedule clock thread (always enabled)
\end{verbatim}

Figure 4.10: Scheduling a Prism system to use rate monotonicity and prune frame overruns

give the clock-tick thread the lowest priority; it will never run and increment the clock while any event dispatch remains to be done. This forces all pending event dispatch to occur before the next clock tick (and hence the next round of timeouts, since they are tightly coupled to the clock updates).
Chapter 5

Experimental Results

In order to judge the effectiveness of the automatic transition system extraction presented in this report, we compare the resource requirements and state space sizes of generated BIR models against their hand-crafted counterparts from [7]. The Prism scenarios tested range from the trivially small (3 components with only one rategroup) to a system which borders on realistic (50 components, multiple rategroups, and several event correlators).

Our trial runs were conducted on an AMD Opteron system clocked at 1.6 GHz with 16 GB of memory. Bogor was run on the Java 2 Platform. Note that the experimental data for the hand-crafted models is reused from its original presentation; we were unable to recreate the codebase as present at the time of its original presentation. In the original simulations, Bogor ran on a Pentium 4 workstation clocked at 2.53 GHz with 1.5 GB of memory. Although this renders wall-clock running time comparisons less insightful, it does not affect the comparisons of the state-space size and the relative slowdowns as model complexity grows—Bogor’s algorithms are deterministic and do not change when a different Java Virtual Machine is used.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Components</th>
<th>Correlators</th>
<th>Rate Groups</th>
</tr>
</thead>
<tbody>
<tr>
<td>BasicSP</td>
<td>3</td>
<td>0</td>
<td>40 Hz</td>
</tr>
<tr>
<td>MultirateSP</td>
<td>6</td>
<td>0</td>
<td>1 Hz, 5 Hz, 20 Hz, 40 Hz</td>
</tr>
<tr>
<td>ModalSP</td>
<td>8</td>
<td>2</td>
<td>20 Hz, 40 Hz</td>
</tr>
<tr>
<td>MediumSP</td>
<td>50</td>
<td>8</td>
<td>1 Hz, 20 Hz</td>
</tr>
</tbody>
</table>

Table 5.1: Test case scenario configurations
5.1 Comparison

If Table 5.2 we see the comparison of state space sizes for hand-coded models versus automatically extracted models. One notices immediately that the state space size as measured by the number of transitions executed is dramatically higher for automatically compiled models in the first two cases (ModalSP and MultirateSP). While this is somewhat surprising, we believe this reflects the more conservative amount of optimization which a general-case translator can do. The hand-coded models (as presented in [7]) were especially optimized in the case of trivially small systems; for instance, most interface methods’ BIR implementations consisted of only one or two invisible transitions; this acted to keep the state space quite small. In comparison, the more general translator is forced to adopt a systematic way of encoding CPS expressions and statements into BIR; this looks clumsy compared to hand-optimized code. The proper analogy to make here is a comparison between unoptimized code emitted by a C compiler and hand-written ASM. The number of possible system interleavings grows linearly with the number of possible thread program counter permutations, so it is not surprising that a small loss of optimization in CPS transliteration causes a state-space blowup factor of 10 or more in small models.

Additionally, it is reasonable to see a larger number of system transitions because the code generated by the new translator does more. All models constructed have a completely faithful encoding of the CPS specification. The previous hand-built models did not (indeed, CPS did not even exist in its present form at that time). Specifically, the newer models use two levels of indirection with invoke instructions to evaluate receptacle method calls; the previous models used no indirection.

It is interesting to note that in MediumSP (the largest available Prism scenario), the automatically-compiled model has a smaller state-space size as measured by number of transitions. We believe this is likely the result of the superior implementation of event correlators; whereas the previous versions were ad hoc combinations of BIR variables and functions, there is now a correlator primitive which performs all internal state changes and event emission in a single atomic step. MediumSP uses several large event correlators, so one expects a commensurate boost in system compactness.

Figure 5.1 illustrates an important point: as the size of the scenario grows, both the hand-coded and automatically translated BIR systems experience about the same asymptotic growth in state space complexity. If this trend holds (MediumSP is currently the largest scenario available) for larger, realistic systems, then the penalty for making model checking of Prism configurations easily
<table>
<thead>
<tr>
<th>Scenario</th>
<th>Transitions</th>
<th>States</th>
<th>Memory</th>
<th>Time (mm:ss:dd)*</th>
</tr>
</thead>
<tbody>
<tr>
<td>BasicSP</td>
<td>Hand-crafted</td>
<td>44</td>
<td>14</td>
<td>0.51 MB</td>
</tr>
<tr>
<td></td>
<td>Compiled</td>
<td>2711</td>
<td>1725</td>
<td>0.28 MB</td>
</tr>
<tr>
<td>MultirateSP</td>
<td>Hand-crafted</td>
<td>150</td>
<td>33</td>
<td>0.61 MB</td>
</tr>
<tr>
<td></td>
<td>Compiled</td>
<td>4036</td>
<td>2566</td>
<td>0.51 MB</td>
</tr>
<tr>
<td>ModalSP</td>
<td>Hand-crafted</td>
<td>6,270</td>
<td>1,560</td>
<td>1.45 MB</td>
</tr>
<tr>
<td></td>
<td>Compiled</td>
<td>22,045</td>
<td>13,596</td>
<td>3.12 MB</td>
</tr>
<tr>
<td>MediumSP</td>
<td>Hand-crafted</td>
<td>360,000</td>
<td>74,500</td>
<td>21.5 MB</td>
</tr>
<tr>
<td></td>
<td>Compiled</td>
<td>329,813</td>
<td>225,658</td>
<td>80.81 MB</td>
</tr>
</tbody>
</table>

*As described in the introduction to this section, the execution time and memory usage for hand-crafted and compiled models are from different machines. They are included merely for reference.

Table 5.2: Experimental data

available is not anything more than a constant “sunk cost” factor.
Figure 5.1: Comparison of resource requirements
Chapter 6

Summary

This report has described the workings of a procedure to automatically extract finite-state transition system descriptions of distributed real-time embedded (DRE) systems. Eliminating the tedious task of constructing finite-state models by hand will make model checking accessible to DRE engineers on Boeing’s Prism product line of avionics systems.

By abstracting away computational details that are irrelevant to the logical modal behaviors of components in a system, we are able to produce a BIR model that—when combined with domain-specific scheduling insights—can efficiently be explored. The preliminary results show that when the translator is used to produce models of realistically sized systems, the state space is only bloated by a small constant factor (2 to 3 times, in our observed results).

In conjunction with ongoing work in Cadena, system designers will be able to give high-level temporal specifications which should hold over the modal behavior of components. The end result is to empower real-world engineers to find subtle logical errors which arise only infrequently because of nondeterminism.
Bibliography


