Using a .NET Checkability Profile to Limit Interactions between Embedded Controllers

David J. Greaves[*], Daniel Gordon,
Atif Alvi[†], Tope Omitola
Computer Laboratory, University of Cambridge

Abstract

Within a closed domain—such as a railway train, chemical production line, vehicle or home of the future—concurrent applications running in embedded controller units (ECUs) and on servers share many common sensors, actuators and feedback paths through the physical part of the domain, while having to abide by common, basic liveness and consistency rules to ensure proper operation of that domain. This paper suggests that all ECUs must export a summary of their behaviour using a restricted subset of .NET bytecode and that the programming constructs used by all participating controllers must abide within a common upper bound so that automated formal checking of domain as a whole is possible. The upper bound is defined as a Checkability Profile. *We describe the ROM and RAM costs of implementing this approach in one of our prototypes: a CD/DVD player for the home of the future.*

Keywords: CLR, .NET, Pervasive Computing, Incremental Model Checking, Application Digest, Feature Interaction.

1 Introduction

CIL (Common Intermediate Language) bytecode is used in the Mono and .NET systems (`www.mono-project.com`) and has been standardised (ECMA-334). A wide variety of compilers exist and the resulting binary files may be run on any CLR (Common Language Runtime) execution platform. In this paper, we propose to use CIL to reflect the behaviour of embedded firmware so that interactions between embedded controllers may be predicted. We also show how to embed assertions in the bytecode that must hold when a device joins a community and which can ensure safe operation under network failure.

Prior work in automated checking of co-operating controllers has frequently assumed that the population of controllers is static and that sufficient system testing or formal checking is applied before the system goes live. In a future of pervasive computing, the population of concurrent applications that wish to share a common pool of sensors and actuators will be dynamic. The example used later in this paper is of an audio-visual and HiFi system in a home, where devices containing embedded applications may be added or removed at anytime. A technology is required that enables real-time or near real-time checking of the system safety and liveness properties so that little or no delay is introduced to the system evolution.

Formal proof methods require that the complete system, including sensors, actuators, applications and real-world feedback effects, is modelled using a common formalism. Unfortunately, there is no widely-accepted or widely-applicable formalism that is amenable to automated reasoning. Indeed, there is a trade off between expressibility and checkability in the selection of a formalism, because when the language is more expressible, undecidability is introduced. There are two fundamental dimensions to the complexity of a computer program: the formal model of the code and the amount of the code. Standard formal models include finite-state, pushdown, linear, octagon, Peano and Turing complete. Measures for the amount of code have less rigorous mathematical definitions, but one can count the number of instructions in a given language, or number of states and transitions or growth order, and so on. Given a system description in one of these formalisms and some safety and liveness checks that need to be dispatched, one must manually select the best combination of formal proof methods, such as model checking, predicate abstraction, symbolic trajectory analysis and automated theorem proving. It is not our aim in this paper to define actual checkability profiles. Instead, we use only one, namely the finite state profile, along with a framework that can encompass other profiles in the future. We envisage that all participating embedded and server-based applications in a domain will conform to a common checkability profile and hence be checkable, provided an appropriate checking server is associated with the domain. This approach is similar to code

[*]David.Greaves@cl.cam.ac.uk
[†]Supported by the Higher Education Commission of Pakistan.

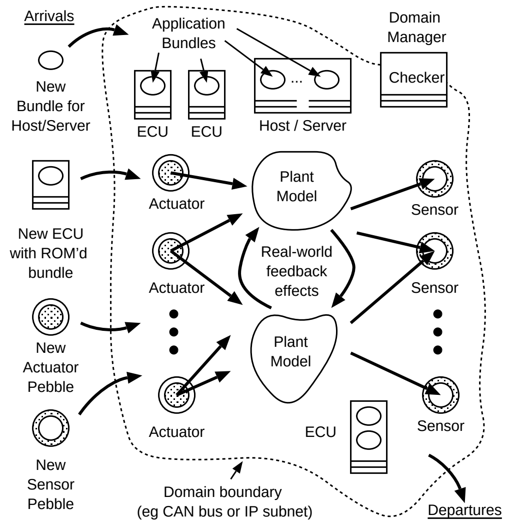

Figure 1. Domain of Sensors, Actuators and Applications under the scruitiny of the Checker on the Domain Controller.

profiles currently used in safety-critical systems for Ada and C++ [1, 7] but differs because we concentrate on the formalism needed to execute the programs rather than the syntactic constructs used to write them.

In general terms, we expect profiles to be arranged in a partial order, with a given automated checker at the domain manager being able to check all programs below and including a given complexity. This assumption motivates the use of CIL bytecode as the description language. Given the relatively long lifetime of certain hardware devices, such as control valves and television sets, we expect checking capabilities to grow greatly while they are deployed. Using a scale of complexity profiles over a full language, such as CIL bytecode, we make the system future-proof, allowing future application programs to get arbitrarily close to being undecidable, while still checkable when connected to old equipment.

2 Proposed Architecture

In our approach, space is divided into domains and devices are prevented from sending commands over the network in a domain until their internal application(s), the *canned application bundle(s)*, has/have been validated by a domain manager. In order to do this, they offer an *application digest* which is a description of their active behaviour, so that automated reasoning techniques can be run before granting a bundle the right to send commands. There are many potential forms of application digest: in this work we investigate the costs of reflecting the key behaviour of the device using CIL (.NET) bytecode, lightly embedded in XML, over HTTP.

As illustrated in Figure 1, a domain corresponds to a community of participating applications that are sharing resources and which may interact in various intended and unintended ways. A simple domain is typically a physical space, such as a house, vehicle or factory. We have considered the need for nested and dynamic domain boundaries, but that is beyond the scope of this paper. The manager uses a summary of the behaviour of each component to check whether the components are compatible. Each component may also offer assertions that it wishes to be held in any domain it joins. Finally, there are a number of standing rules of each domain, including one that prohibits command conflicts. In a basic approach, devices join a domain using a first-come, first-served discipline, where assertions over domain behaviour introduced by one device may prohibit the joining of a subsequent device. Using priorities and ejection, the dependence on arrival order can be arbitrarily reduced, but further discussion of that is also beyond the scope of this paper.

To make effective use of today's automated reasoners, such as model checkers, we must reduce the complexity of applications and write them in a stylised way that conforms to a target checkability class. We use an architecture where, as far as possible, networking complexity is served from a standard library, called *tuplecore* and the I/O devices contain as much functionality as possible. The I/O devices are called *pebbles*. They include all sensors and actuators. We use the term *application bundle* to denote an application written in this way. The term was used because our current checking technology converts the application to a list of concurrent, finite-state rules.

To fit within our initial, finite state, checkability class, bundles additionally posses the following properties:

1. All network naming and binding operations are performed by rewriting the bundle before launching it.
2. All variables have a discrete range.
3. All dynamic storage allocation is done in a start-up phase, before a main process loop is entered (this includes forking threads).
4. All array sizes and heap and stack use are determined before the main process loop of a thread.

However, we envisage these restrictions will be relaxed in future checkability classes.

Pebbles themselves are self-contained hardware or software objects that fulfil a certain task. A Pebble can represent a fairly large chunk of functionality: it could be a hardware component, such as a wall-mounted keypad or a fire alarm klaxon, or it could be an entirely virtual component hosted somewhere on the network, such as a speech recogniser. Pebbles do not invoke operations on or interact with

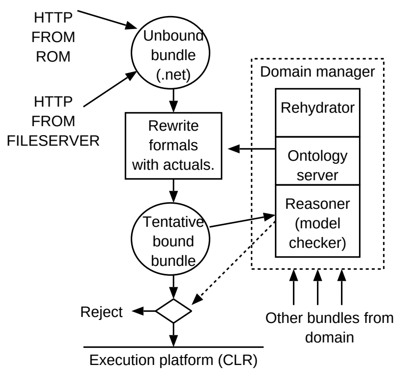

Figure 2. Rehydration of a bundle.

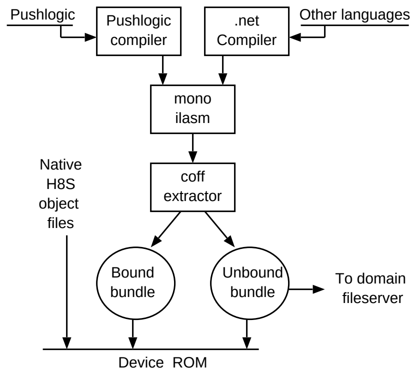

Figure 3. Tool Flow.

other pebbles—interaction is left to bundles. Pebbles are like a combination of a device and its device driver. An application digest may be exposed by a pebble to describe its internal reactive behaviour or known effects on other pebbles.

Application code supplied in the ROM as part of a device may already be bound to control the local pebbles inside that device. Other bundles may be supplied in *unbound* form, ready to be hosted on servers in the domain should they be needed. We use the term *rehydration* to denote the process of taking a bundle of code, rewriting formal device addresses in the code with bindings of actual devices, and firing up the rewritten bundle on a server (Figure 2). The code that implements dynamic binding is therefore an 'out-of-band' system-wide service and so not checked on the fly. The bundle itself is consequently simplified and much easier to check. An unbound bundle might be rehydrated several times, each time with different bindings, to control a multitude of similar pebbles in a domain.

Each ECU contains a network interface, TCP/IP stack and a tiny web server. The webserver serves XML and canned stylesheets for when the XML is viewed inside a browser. The webserver could also export pictures for visual user interfaces. Platforms beacon every few seconds using an XML message containing basic device information, sent on the local UDP broadcast address (or equivalent mechansim on the CAN fieldbus). These infrastructure subsystems provide the basic facilities of UPnP to enable device location, registration and description of internal pebbles, bundles and infrastructure components [6]. Beaconed information is stored as soft-state in the domain manager using the Protégé ontology server (`http://protege.stanford.edu/`) and deduction rules in the ontology server can trigger rehydration. For example, presence of a smoke sensor and a klaxon can start a fire alarm service.

Adding this level of functionality to the lowest-level components in a system is questioned on cost grounds. However, studies have shown that the major cost is XML parsing [5] which is avoided in our solution, where code running from low-level ROM only emits XML. We report the full costs for an actual device in § 4.

3 Reflecting the Embedded Code

Although we advocate reflecting the behaviour of bundles using CIL bytecode, we retain freedom of implementation within the actual ECU. It can either execute the exact same bytecode that is reflect as the application digest, or it can execute a binary/JIT version of the same, or it can execute a more elaborate program that bi-simulates the reflected program in terms of externally visible activity.

In our practical work so far, the devices export their CIL code verbatim as their application digest (Figure 3). CIL executable code is relatively concise and its metadata table structure is also highly compact. The metadata consists of symbol tables containing definitions of fields, types, methods, classes, custom attributes and so on. The digest is exported over the network using HTTP GET commands to the embedded web server. The exported files are an XML encoding of the CIL `.text` region from their coff file format and the XML is generated on-the-fly from the canned binary image of the bundle. The pebble descriptions are also XML strings and, likewise, are generated on the fly from a ROM data structure.

When a CIL bundle starts, static fields occurring in the bytecode that are flagged with CIL *custom attributes* are mapped to tuplecore variables for network I/O. Custom attributes are the defined way of extending CIL. They allow embedding of non-functional information. The attributes are revealed during code reflection but during runtime execution the code itself is structured to always obey any constraints designated by the attributes. We use attributes to denote the allowable range of variables and to describe the default behaviour of components when network disconnection occurs, so that we can reason about fault conditions. Custom attributes are also used for thread sensitivity information to specify which threads to unblock on which tuplecore events (button pressed, packet arrived and so on).

Assertions are either summaries of the current bundle or else integrity requests about the domain behaviour that must be met before the bundle (or entire device/service) joins the domain. A number of *system standing assertions* are defined for all domains prior to system start-up. These are sufficient to avoid feature interaction. Assertions are encoded as methods that are not necessarily called from the main thread or any thread forked from it. The simplest assertion method returns a non-blocking 0 or 1 for fail/pass. This is suitable for safety predicates. If desired, they can be turned into run-time monitors using a simple library. Liveness assertions can take this form too but can only be checked with static tools. Assertions with a temporal component to them need to block on some event that the library or reasoner can detect and treat as a next state operator. A dummy `next()` function is provided for that purpose. Temporal path quantifiers are likewise coded as higher-order functions that take their own type as an argument.

4 CD/DVD Player Prototype

We have constructed a number of prototypes for a home of the future. Here we report on the software costs associated with implementing our approach in a CD/DVD player.

The player has no audio or video output facility of its own. Instead, it routes its output through back-panel AV and Ethernet sockets. It has two physical buttons on its front panel that direct its output over the Ethernet. The buttons are permanently labelled with room names: kitchen and lounge. Suppose there is a second such player, perhaps elsewhere in the house. There is a clear possibility that both devices can be asked to send their media to a common output device that might only be able to handle one stream at a time. This sort of problem is commonly called a *feature interaction* and it frequently arises, in one form or another, in pervasive computing environments. Feature interactions are manifest either as oscillations or violations of safety and/or liveness conditions [2]. The case of two conflicting media streams directed at one target is a safety issue. A standing safety condition of the domain is that there are no conflicts between applications. Therefore, if a second application counter-commands a first, the first should be amenable to this situation in terms of its programmed behaviour, i.e. it must have a way of backtracking. For instance, it might compensate by stopping to send its stream. For the player, a liveness rule might dictate that there is always some route to opening the media drawer; otherwise the owner may never get his disc back.

Our prototype was constructed using the box, power supply and front panel from a commercial CD player but with the mechanism and main circuit board replaced using an ATAPI DVD drive and a 10 MHz Hitachi H8S processor with NE2K Ethernet port. We had 1 Mbyte of RAM on our processor card, but a commercial product needs to use as little RAM as possible to minimise silicon area. Mask ROM is generally far less expensive than RAM and our ROM use, shown in Table 1, is not considerable. (ROM code footprints for windows CE are around 0.5 to 1.0 Mbyte (msdn2.microsoft.com/en-us/library/aa913762.aspx)).

The player uses *miniclr*, a cut-down common language runtime we developed that executes CIL bytecode where the support for each possible .NET instruction can be conditionally included or excluded from the machine. A static scan of the bytecode determines which of the 208 CLR instructions are needed in the ROM image for the player. Most of the real work was implemented in native C code in the tuplecore and the pebbles and so only 25 different instructions are needed for the player. It has to support only static fields, integer arithmetic, boolean connectives and comparison of enumeration types. Missing instructions include those for handling floating point, pointers, exceptions and objects. No garbage collection is required since there is no dynamic storage allocation (beyond bundle start-up). The bytecode was generated with the SPL1 Pushlogic compiler [3] that guarantees finite-state code and which also generates default code to handle execution conflicts between different bundles, performing what it calls a *pushback*.

As illustrated in Figure 4, the player contains the following architectural components:

- **Display Interface Pebble** A VFD display for hours, minutes, seconds and miscellaneous status.
- **Keypad Pebble** Thirteen push-to-make keys.
- **Timer Pebble** A standard device present on all AutoHAN execution platforms.
- **Mechanism Pebble** The CD/DVD mechanism.
- **Bound Bundle** CIL bytecode bundle: already checked.
- **Unbound Bundle** CIL bytecode bundle for use once authenticated.
- **Infrastructure** Network interface, tuplecore, miniweb web server, miniclr bytecode interpreter.

The architectural components of the device have only two things in common: first, that they are physically implemented inside the same casing and so share common resources, such as the microprocessor and power supply, and second, that they are joined together to function as a whole by the bindings of the canned application.

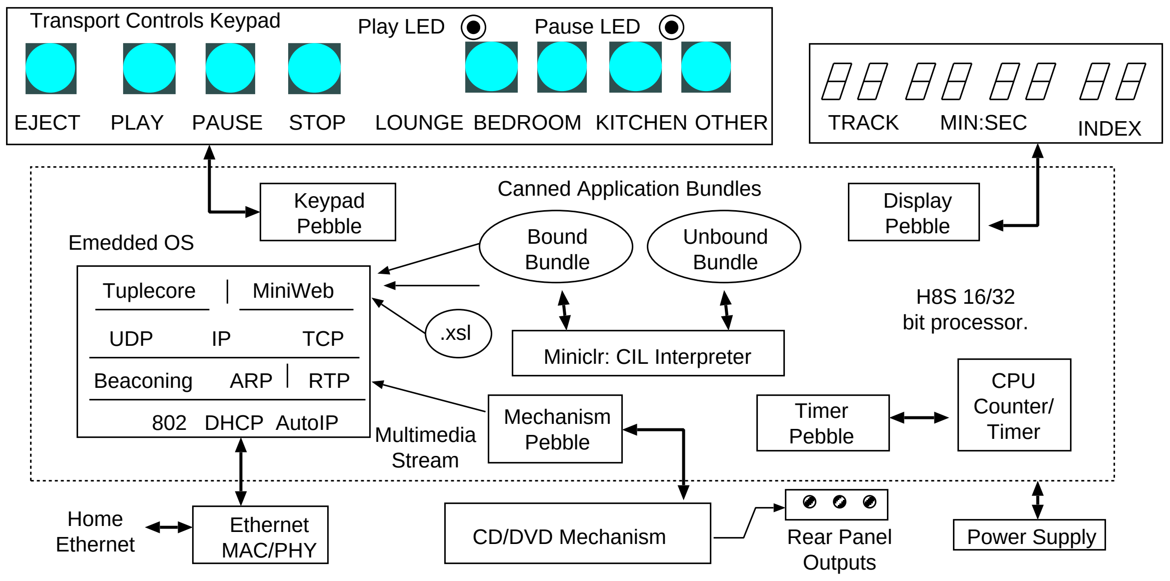

Figure 4. CD/DVD Player Block Diagram

When the device is switched on, each non-infrastructure component beacons on the network interface and registers itself with network services. The bound application contains hardwired addresses for the local pebbles and can thus commence execution without contacting the domain manager over the network. It is authorised to do so since it has no pro-active behaviour over the network on external pebbles. The pro-active behaviour of the device is to send media streams to the labelled designations, lounge and kitchen. These operations are not allowed until after the device has been checked for feature interaction by the domain checker. In addition, these functions require binding of the buttons to their targets. Therefore, the unbound bundle implements the additional functions once rehydrated. As shown in Figure 2, it is fetched by the domain manager, rewritten with device bindings and then run on any suitable host, provided it is compatible with the other bundles, their constraints and the standing constraints of the domain. The player itself could potentially run this bundle as well, once rewritten, but we instead run it on another CLR at the domain manager. The tuplecore library gives sufficient location transparency for the same bundle bindings to be used in either location.

Table 1 shows the ROM and statically allocated RAM use for each software component of the player. The TCP/IP implementation was `lwip` from SICS configured to use 16 pbufs and 8 TCP connections. Since tuplecore and beaconing use UDP, the maximum number of TCP connections needed for the infrastructure is just 1, assuming one domain manager is reading just one reflected segment at once. When a web browser is also directed at the player, it can use two connections because it may suspend XML fetching while getting an embedded stylesheet.

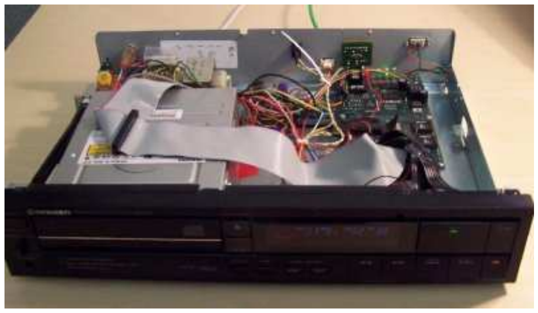

Figure 5. Prototype DVD/CD Player.

The heap space used was tracked by annotating the memory manager to log active use on a serial port and this is plotted in Figure 6 as the player booted, beaconed and had its canned files read out by the domain manager. A total of up to 5 KB was used. 3 KB was used on the stack. The largest RAM consumer was the Ethernet packet buffers, and one-third of this was devoted entirely to handling IP fragmentation, hence commonly wasted. Overall, we predict our system will comfortably run on a processor with 32K RAM, which is less than is commonly devoted just to the anti-jog buffer in a portable CD player.

5 Checking Procedure

Although we are advocating an architecture for generic checkability, in this section we demonstrate a specific checking result for the player. A number of proof tools

Name	Description	ROM		RAM
crt.o	C runtime system	396	0.3%	0
Pioneer.o	Main,Keypad Pebble, Upcalls	6832	5.8%	40
libc.o	C library	3340	2.8%	16
socket.o	Sockets library	4572	3.9%	4
threads.o	Thread library	1680	1.4%	8
cdrom.o	Mechanism Pebble	4382	3.7%	4
timer.o	Timer Pebble	260	0.2%	0
isapnp.o	Address binding	4824	4.1%	0
pbuf.o	Buffer handling	2586	2.2%	3316
tcpudp.o	TCP + UDP protocols	19023	16.3%	18
netif.o	Layer 2 protocols	1478	1.2%	12
mem.o	Memory allocator	1814	1.5%	Fig 6
ip.o	IP protocol	5526	4.7%	7396
arp.o	ARP protocol	3342	2.8%	4
ne2k.o	Ethernet driver	4200	3.6%	26
autoip.o	Auto IP protocol	1004	0.8%	0
libcio.o	RS232 debug port	466	0.4%	0
vdf.o	Vacuum Display Pebble	1126	0.9%	2
tuplecore.o	Tuplecore library	18473	15.8%	1176
ramfs.o	Web server and filesystem	10508	9.0%	4
bcreflect.c	Bundle reflection	3476	2.9%	0
miniclr.o	CLR interpreter	16930	14.5%	12
Subtotal machine code + RAM total		116238	100%	
pioneer_canned.o	IL bytecode	26052	0	
	Canned html and stylesheets	4384		
miniclr.o	CLR opcode tables	2052		
Total ROM Code		148726		

Table 1. ROM and RAM Memory Use Breakdown (excluding stack)

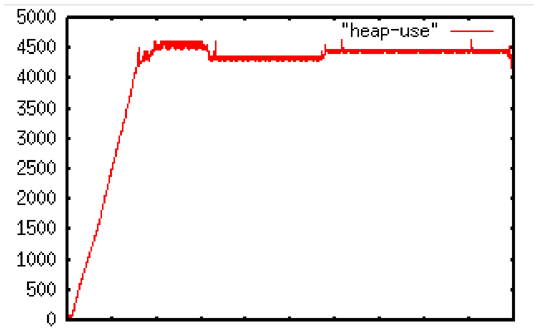

Figure 6. Player Heap Memory versus Time (bytes).

specifically exist for checking CIL bytecode, but we used software that converts CIL programs to hardware circuits and then checked the resulting circuits.

Formal methods applied to digital electronics have been successful at checking large finite state machines. In addition, there has been much study of generating hardware designs from software programs, including CIL programs generated from C# [4, 8]. These algorithms attempt to reduce arbitrary software code to hardware finite-state machines. The basic techniques require that the code sequence behind the main entry point consists of a linear or unwindable, non-blocking, start-up path that forks all other threads, performs all binding operations and allocates all static arrays and all of the dynamic storage that is going to be used, followed by an infinite process loop which implements a labelled transition system (LTS).

Our domain manager collects the application digests, in CLR form, from every ECU and server, converts them each to LTS form and then performs input chain and cone-of-influence reductions. These reductions greatly minimise the aggregate LTS size. They find certain sub-machines that can be replaced with unspecified inputs and by removing anything that cannot have any effect on the proof goals.

Execution of CIL constructor '`.ctor`' methods is included in the start-up path. The start-up path is split off from the main process loop by running an interpreter on the code up to the first blocking primitive or to the top of the first loop that cannot be unwound (for reasons of it being infinite or the number of trips depending on run-time data values). The main process loop of each thread is converted to LTS form by defining a sequencer. The sequencer is an automaton with a state defined for each blocking primitive and for each .NET program label that is the destination for more than one flow of control. The I/O operations and operations on local variables performed by the program are all recorded against the appropriate arcs of the sequencer.

The bound bundle from the player contains 9539 CIL instructions. However, these are all deleted by cone-of-influence and input chain trimming. The unbound bundle contains only 72 CIL instructions, many of which are only executed when the bundle is hydrated. The unbound bundle has a state space of just 27 states. This bundle was read by the domain manager and replicated 99 times to generate an artificial scenario containing 100 players. The domain manager currently uses finite-state model checking using either Spin, NuSMV or its own BDD package. With NuSMV version 2.3.3 on a 1.3 GHz linux platform, a conflict between the bundles was found in 5.8 seconds using just over a million BDD nodes. Arithmetic operations in the bound bundle are only used to increment and decrement track numbers, and hence the bundle may be classified as linear. A tighter classification is finite-state, since the track numbers are limited to the range 0 to 99. In future work we will experiment with other checkers and perhaps define corresponding checkability classes.

6 Conclusion

The API to our networked DVD/CD player is much like that for previous DVD/CD players from HomeAPI, UPnP or HAVi, but we have restricted the XML parsing to save RAM costs. The overheads associated with the other non-binary coded protocols, such as HTTP, are clearly not prohibitive. Indeed, complete webservers are reported running in 64K address maps (`www.d116.com`). Implementing full UPnP with XML parsing is probably too complex for cheap ECUs [5, 9], so we avoided it.

High-integrity systems that are correct by construction are an alternative, such as using Pacc (`www.sei.cmu.edu/pacc`), but these cannot handle dynamic and ad hoc systems. Unlike approaches that effectively define a syntactic subset of a given language, the use of CIL allows any language which has a suitable compiler to be used. Proof carrying code enables a particular proof to be reproduced at a later date, whereas our approach restricts the code such that a large number of automated proofs are readily possible. Whereas previous approaches have required system evolution to be restricted and integrity to be maintained by non-electronic means, our approach supports fully automatic maintenance of system evolution.

The tuplecore and pebble abstractions allow a complex, distributed system containing complex software to be viewed in potentially simple terms, with correspondingly simplified application bundles. We are able to expose the essential behaviour of the system for automated reasoning without getting side-tracked with complex networking and binding code.

Because rehydration is controlled using formal rules held in an ontology server, we can reason in advance about certain evolution steps, thereby reducing the amount of real-time checking needed,

Furture work will look at precomputing information from application digests to assist with rapid incremental model checking and assume/guarantee reasoning, since for large systems we know that LTS model checking will become unacceptably slow.

Although we do not need the whole CLR on the player, a motivation for reflecting the code using a standardised bytecode is that additional features may be needed on other execution platforms and code reflected by all such platforms can be checked within a common framework at the domain manager. The embedded code in our examples was .NET/-Mono CIL code generated from a special compiler that ensures the structure of it will meet our first generation checking requirements, which are finite state. However, we envisage that our technique can be generalised in the future by relaxing the compiler constraints or enhancing the checker capability. In other words, we envisage various *checkability classes* to be defined, as automated checking capabilities develop, with devices using increasingly broader forms and structures of bytecode being acceptable to classes defined later in time.

References

[1] Ada Rapporteur Group. Ravenscar profile for high-integrity systems. ISO/WG9 AI95-00249. Technical report, Ada Rapporteur Group, 1995.

[2] M. Calder, M. Kolberg, E. H. Magill, D. Marples, and S. Reiff-Marganiec. Hybrid solutions to the feature interaction problem. In D. Amyot and L. Logrippo, editors, *FIW*, pages 295–312. IOS Press, 2003.

[3] D. Greaves and D. Gordon. Using simple Pushlogic. In *WEBIST 06:Proceedings of the second international conference on web information systems and technologies*, 2006.

[4] E. Grimpe and F. Oppenheimer. Extending the systemc synthesis subset by object-oriented features. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 25–30, New York, NY, USA, 2003. ACM.

[5] H. Hayakawa, T. Koita, K. Sato, and A. Fukuda. Adaptation of sonica for p2p architecture. In *IMSA'07: IASTED European Conference on Proceedings of the IASTED European Conference*, pages 82–87, Anaheim, CA, USA, 2007. ACTA Press.

[6] Microsoft. Universal plug and play device architecture, version 1.0. Technical report, Microsoft, 2000.

[7] MISRA. Misra: Development guidelines for vehicle based software. MIRACV10 0TU. Technical report, MISRA, 1994.

[8] S. Singh and D. Greaves. Describing hardware with parallel programs. In *Designing Correct Circuits*, 2008.

[9] T. Tranmanh, L. M. G. Feijs, and J. J. Lukkien. Implementation and validation of upnp for embedded systems in a home networking environment. In *Communications, Internet, and Information Technology*, pages 279–284, 2002.