Defining Checkability Classes for CIL Bytecode Extended Abstract

David Greaves

University of Cambridge, Computer Laboratory

Standard CIL bytecode is used in the Mono and .net systems [ECMA-334]. We use CIL to reflect the behaviour of embedded firmware so that interactions between pervasive computing devices may be formally verified. We also embed assertions in the bytecode that must hold when a device joins a community and which can ensure safe operation under network failure.

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: we investigate the costs of reflecting the key behaviour of the device using CIL (.net) bytecode, lightly embedded in XML, over HTTP. Accompanying the program is a classification of the *complexity* of the program. In this paper, the complexity classification we are using is that the program describes a finite state machine, and hence is of a coding style that can be readily converted to a hardware logic design by certain toolchains used in design automation. In general terms, we expect complexities 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 provide futureproofness.

Our checker covers the major forms of network-level error, which arise from lost packets, network segmentation and device disconnections. A related refinement is network latency: where a safety condition is a function of conditions at different locations, we can check consistency predicates of the make-before-break and break-before-make form.

Our work contrasts with work that proves the correctness of individual network protocols in isolation from the application behaviour. Our approach spots feature and deadlock interactions between actual applications and is then refined to encompass networking protocol errors. Although we have performed checks by essentially converting the aggregate collection of applications into a global LTS, reflection using CIL is sufficiently general that other forms of automated reasoning are not precluded.

1 Further Details

Suppose I have a CD/DVD player with a network connection. The player has no audio or video output facility of its own. Instead, it has a number, say four, of physical buttons on its front panel or remote controller that direct its output to networked devices. The buttons are permanently labelled with several room names, such as: lounge, bedroom, kitchen, other. Now suppose I have a second such player, perhaps elsewhere in the house. There is a clear possibility that both devices will be asked to send their media to a common output device, whereas that target might only be able to handle one stream at a time. This sort of problem is commonly called a *feature interaction* and it arises in any pervasive computing environment, where a potentially disparate collection of application programs attempt to use the same actuators or output devices. Feature interactions are manifest either as oscillations or violations of safety and/or liveness conditions The case of two conflicting media streams directed at one target is classed as a safety issue. A standing safety condition of the domain is that there are no conflicts between applications, so if a second application counter-commands a first, the first *MUST* be amenable to this situation in terms of its programmed behaviour: i.e. it must have a way of backtracking or *compensating*: for instance, it might stop sending its stream. For the player, a liveness rule might dictate that there is always some route to opening the media draw, otherwise the owner may never get his disc back.

Although we do not need the whole CLR on any one device, 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, but 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.