

# Improving TinyOS Developer Productivity with Statecharts

Volker Menrad, Miguel Garcia, Sibylle Schupp

Institute for Software Systems  
Hamburg University of Technology

{menrad, miguel.garcia, schupp}@tu-harburg.de

**Abstract**—The development of sensor network software is challenging due to (a) the emergence at runtime of non-intuitive component interactions; and (b) the difficulty of debugging non-reproducible bugs, especially those arising from timing and scheduling conditions. These features are inherent to the TinyOS programming model (which involves wiring off-the-shelf and custom components, to communicate over interfaces) and thus approaches are needed to increase both productivity and quality measures of sensornet development. The state of the practice involves documenting interface contracts in natural language, so as to write components that fulfill the expectations of the invoked component (its preconditions), upon which a guarantee is obtained about the resulting component state (its postconditions). Experience has shown that this kind of technical documentation leaves room for misunderstanding, and thus a more precise format is required to capture these contracts. Using real-world examples we show how a rich statechart language (a) can be used to achieve readable yet more precise formulations of interface contracts; (b) in the future can also serve as a basis for automated verification of nesC code (i.e., to determine whether such code abides by the contracts formalized with statecharts). Our case studies build upon previous work on contract specification for component-based software, however to our knowledge this is the first time that an expressive statechart language is systematically used to capture full-fledged TinyOS behavioral contracts, with an eye towards the compile-time verification of contracts.

**Index Terms**—static software analysis, TinyOS, nesC,

## I. INTRODUCTION

TinyOS is a component-based system in the sense that invocations between modules abide by command and event signatures grouped into interfaces. From the perspective of the compiler, these interfaces are purely syntactic: any conforming sequence of invocations will be accepted at compile-time, although its execution might result in a runtime error. In addition to their definition in source code, another source of information about interfaces are the TEPs (TinyOS Enhancement Proposals), which contain, among others, natural language descriptions of disallowed values for arguments, and restrictions on invocation sequences. An invocation to the `send(msg,`

`len)` command in the `Send` interface will compile even if `null` is given as first argument, which is a poor choice because it will invariably lead to a crash. Even worse, the technical documentation of `Send` does not warn against this. Instead, it contains rules on what happens if the maximum message size is exceeded, as well as a number of additional rules that are non-trivial to check manually by the developer.

Even in those cases where TEPs are complete, natural language leads quite easily to misunderstandings due to ambiguous interpretations. As a consequence, different implementors of the same interface may make different assumptions thus compromising portability of client code. The problems sketched above also plagued other component-based systems, leading Bertrand Meyer to put forward *Design by Contract* [1].

## A. WHAT IS DESIGN BY CONTRACT

Design by Contract enriches the syntactic information that interfaces provide by adding three kinds of boolean-valued expressions to each method signature. These expressions capture *preconditions* (i.e., conditions that should hold for an invocation to succeed), *postconditions* (i.e., what is guaranteed upon completion of the invocation), and *invariants* (which state what should not change due to the invocation).

These additional specifications, because of being machine-checkable, constitute a solid basis for new bug detection techniques and dispel doubts and avoid redundancies, resulting in more precise *contracts* between modules. In the original proposal by Meyer, contracts are checked at runtime, as demonstrated for TinyOS by Archer, Levis, and Regehr [2]. Using this technique, once a bug has been detected ingenious ways have to be employed to report it to the outside world, e.g., by means of LEDs to convey the binary address of the offending instruction.

Nowadays, it is recognized that contracts are also amenable to *static verification* techniques, a kind of software analysis to automatically find out at compile-time under which execution traces certain contracts will be broken. One such technique is *model-checking*, which conceptualizes executions as trees of states, with root states derived to cover all initial program inputs. In contrast to runtime checking, which exercises a few code paths, a model-checker performs an exhaustive search (within bounds) of all possible code paths, thus leading to better *coverage* at a lower cost than manual testing in conjunction with runtime checking.

The coverage achieved by the admittedly finite analysis of a model-checker is much higher than testing because states can be manipulated symbolically, and several properties of interest depend on the shape of an object graph rather than on its size or concrete attribute values (the “small scope hypothesis” [3]). For example, the condition “two lines intersect” can be manipulated without considering concrete crossing points. Model-checkers can detect these situations, taking a single state as representative of all those exhibiting such shape.

Concerning contracts for TinyOS interfaces, the focus are pre- and postconditions. An interface defines commands and events, so that an interface user can call commands that were implemented by the interface provider. In turn, the user has to implement handlers for the interface’s events. Contracts in TinyOs are thus bidirectional, meaning that an interface user has to fulfill the preconditions for commands, and the postconditions for events.

## B. EXAMPLE CONTRACT: SEND INTERFACE

The example in Listing 1 (reproduced from [2]) shows contracts for the command `send` and the event `sendDone`, both part of the `Send` interface.

The first precondition already rejects null as packet to send, while its accompanying postcondition captures the memory ownership violation discussed later in this paper and other aspects of the protocol that developers are expected to follow when using this interface, as defined in TEP 116.

In more detail, the code in Listing 1 performs runtime checking of pre- and postconditions. In a precondition, the argument to `assume()` is evaluated, aborting the program in case it evaluates to false. In a postcondition, `guarantee()` plays a similar role. As a whole, the contract requires keeping track of three states for the interface (`SEND_IDLE`, `SEND_SENDING`,

`SEND_CANCELLED`) and two states for each message (`USER_OWNED`, `OS_OWNED`), with transitions between them taking into account the error code returned by `send` (any of `EBUSY`, `SUCCESS`, or `FAIL`).

Internally, a module providing `Send` has a single slot for outgoing packets, which rules out having more than one pending message on the send queue. This explains why `send` cannot be invoked again until a `sendDone` notification has been received from the provider. To enforce this protocol, the pre- and postconditions for the `send` command state when the command can be invoked (i.e., while in states `SEND_IDLE` or `SEND_CANCELLED`) and which new state is in effect after completion of the execution of the command. For example, if the start state is `SEND_IDLE` and the transmission does not succeed, `SEND_IDLE` remains as current state.

```

error_t send(message_t* msg, uint8_t len){
    error_t retVal;
PRE:
    assume(msg != 0x00);
POST:
    guarantee(retVal == SUCCESS ||
              retVal == EBUSY ||
              retVal == FAIL);
    if(send_state == SEND_SENDING)
        guarantee(retVal == EBUSY);
    if(send_state == SEND_IDLE) {
        guarantee(retVal == SUCCESS ||
                  retVal == FAIL);
        if(retVal == SUCCESS){
            msg->msg_state = OS_OWNED;
            send_state = SEND_SENDING;
        }
    } else
        guarantee(retVal == EBUSY);
}

void sendDone(message_t* msg, error_t error){
PRE:
    assume(msg != 0x00);
    assume(send_state == SEND_SENDING ||
           send_state == SEND_CANCELLED);
POST:
    if(send_state == SEND_CANCELLED)
        guarantee(error == ECANCEL);
    else if(send_state == SEND_SENDING)
        guarantee(error == SUCCESS ||
                  error == FAIL);
    send_state = SEND_IDLE;
    msg->msg_state = USER_OWNED;
}

```

Listing 1. `Send`.contract

Each message is a pointer to a structure in memory. Once the `Send.send` command is invoked on this message, this memory area should not be touched. To keep track of the memory ownership of the message, an extra field is appended to the `message_t` structure. When a message is created, it is `USER_OWNED` and through the command `Send.send` the ownership is transferred to the operating system (`OS_OWNED`). When the message was sent, the memory area can now be reused by

the user, therefore the ownership is returned to the user when the `Send.sendDone` event is signaled.

As can be seen from the example, descriptions of state transitions can become lengthy. In order to improve their readability, Harel devised the statecharts notation [4].

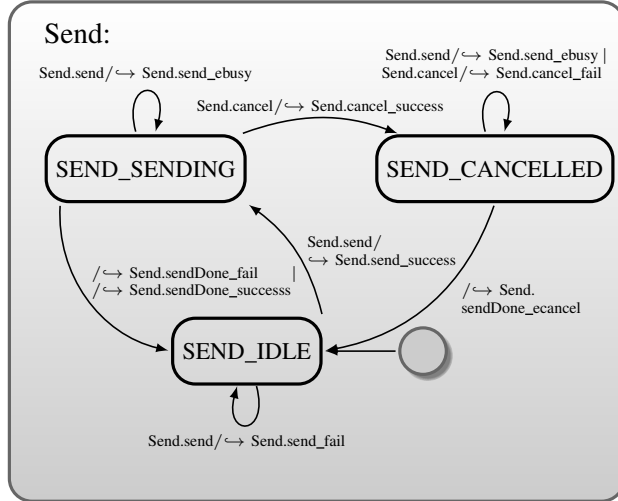


Fig. 1. SendInterfaceStatechart

## II. GRAPHICAL REPRESENTATION OF CONTRACTS

Statecharts [4] improve on the readability of finite state machines by allowing hierarchy on states and parallelism on transitions. In a nutshell, besides plain states and initial pseudo states, so called *composite* states are allowed. A (non-empty) composite state is either a composite *or-state* or a composite *and-state*. Each *or-state* contains a complete statechart and when active one of its contained top states is active. An *and-state* contains two or more parallel *regions*, each of which can again include a whole statechart. A transition is a relation between two states. The label of a transition consists of one event/action sequence. In an event/action sequence, the symbol `/` separates the event and action parts, where the event part names the event that triggers this transition. An empty name indicates implicit reaction to a *completion event*, i.e., this transition is triggered immediately. The action part consists of a possibly empty list of statements, separated by semicolons, which are executed when the transition is taken.

A trace is considered illegal if an event cannot be handled in an active state, i.e., it is an error for an event to be received in a state without any outgoing transitions labeled with it. Conceptually, a

trace comprises the states that are visited and the events happening in between.

In this paper and in order to save space, an edge between two state nodes in a statechart represents *one or more* transitions, where the event/action sequences belonging to each transition are separated by the symbol `|`. A statement in the action part may trigger an event on another statechart, for which the symbol  `$\leftrightarrow$`  is used.

For example, in Figure 1, there is one transition from the `SEND_CANCELLED` state to the state `SEND_IDLE` with an empty name and as action this transition triggers the event `Send.sendDone_ecancel` (`/  $\leftrightarrow$  Send.sendDone_ecancel`).

```

module MeasTempC {
uses {
interface Boot;
interface Send;
interface Timer<TMilli> as MilliTimer;
interface Packet;
interface Read<uint16_t> as TemperatureRead;
interface SplitControl as RadioControl;
}
implementation {
uint16_t meas_temperature;
mmessage_t msg1;

event void Boot.booted() {
call RadioControl.start();
}
event void RadioControl.startDone(...) {
call MilliTimer.startPeriodic(250);
}
event void MilliTimer.fired() {
call TemperatureRead.read();
}
event void TemperatureRead.readDone(...) {
meas_t* meas_payload;
/* here should be checked whether we own msg1
before updating it */
meas_payload = (meas_t*)call Send.getPayload(&
msg1, sizeof(meas_t));
meas_payload->temperature = data;
call Send.send(&msg1, sizeof(meas_t));
}
event void Send.sendDone(...) {
/* here we should record that we've recovered
ownership of msg1 */
}
...
}

```

Listing 2. MeasTempC.nc

## III. CASE STUDY: MEMORY OWNERSHIP VIOLATIONS

The application `MeasTempC` in Listing 2 serves as a simplified example of a typical sensor reading applications. In this case, temperature is periodically measured for sending over `Send`, which relies on the split-interface mechanism for transmitting messages (i.e., invoking the `send()` command returns immediately, and once the transmission of the message completes, a `sendDone()`

event is signaled). Ignoring the comments, the code appears to fulfill its duty. After initializing the hardware and software, which is signaled through the event `Boot.booted()`, the radio is turned on and a timer is installed with the command `MilliTimer.startPeriodic()`. This timer will periodically signal when a reading period starts. In the handler for this timer event the reading is started, whose completion results in another handler, `TemperatureRead.readDone`, being invoked. At this point, a temperature value is ready for transmission, which we record in a structure with the assignment: `meas_payload->temperature = data;` The next statement starts the split-interface for sending over `Send`.

The problem with *MeasTempC* is that the order in which outstanding handlers are invoked cannot be controlled. Provided that readings or transmissions have been started, the following three signals `Send.sendDone()`, `TemperatureRead.readDone()`, and `MilliTimer.fired()` can be received in any order. In particular, two consecutive `readDone()` notifications can be received without an intervening `sendDone()`. At the very least, this timing of signals would lead to losing a temperature reading. Additionally, the pending transmission might transmit a corrupt package, because of *packet ownership* [5, §3.5.1]:

If the interface user modifies the packet after passing it to the interface provider, the packet could be corrupted. For example, the radio stack might compute a checksum over the entire packet, then start sending it out. If the caller modifies the packet after the checksum has been calculated, then the data and checksum won't match up and a receiver will reject the packet.

As can be seen, the application *MeasTempC* may modify a packet (in `readDone()`) whose ownership was acquired by TinyOS in `send()` and not yet released in `sendDone()`. As in any other violations of behavioral interface, the compiler will silently accept the application. In contrast, the model-checking approach presented next detects this erroneous program logic at compile-time.

#### IV. VERIFYING USER PROGRAMS AGAINST CONTRACTS

The key insight to finding the execution trace discussed above is to model all components wired by the user application as a top statechart consisting

of parallel regions within an *and-state*  $T$ . One of these parallel regions stands for the implementation of the main component (the “user application”). Additionally, for every interface used and for every data item manipulated by the user application, there exists a region in  $T$ , representing its status. Interaction across parallel regions is possible: an event triggered within one parallel region may cause one or more transitions to be taken in other region(s).

#### A. ERRONEOUS TRACE

The statechart in Figure 3 models the user application `MeasTempC.nc`. In the Figure, states represent points in the execution of *MeasTempC*.

The initialization of hardware (e.g., Radio) and software (e.g., Timer) is modeled through the first four states (`INIT HW`, `WAIT`, `RADIO`, `INIT DONE`). To make the diagram less cluttered, the return values `EBUSY` and `FAIL` of the command `RadioControl.start` have been omitted.

After initialization, application behavior differs, before and after the first message transmission request has been successfully accepted. Accordingly, two loops appear in Figure 3, differing only in the self loop (in `MAIN` respectively `NO SUCCESS`). The first loop consists of the states `TIMER`, `READ`, `SEND` and `NO SUCCESS`. On the other hand, the so called main loop consists of the remaining states. In detail, during the main loop, the `Send.sendDone` event (with the error value `SUCCESS`) can appear, unlike in the first loop.

The erroneous event sequence leading to an ownership violation can be summarized as follows: (a) after arriving at state `MAIN` in region `MeasTempC`, the message just sent has

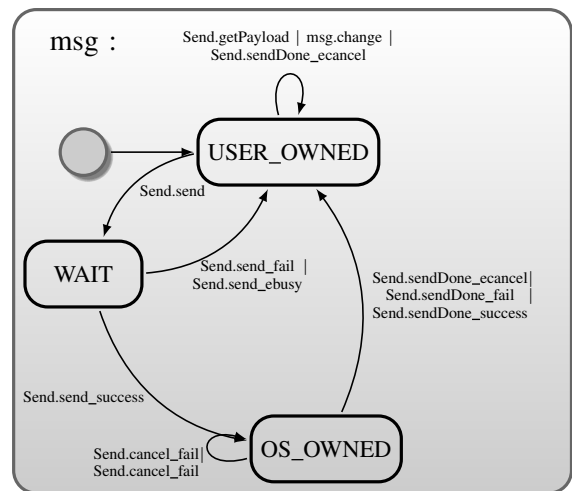


Fig. 2. MessageStatechart

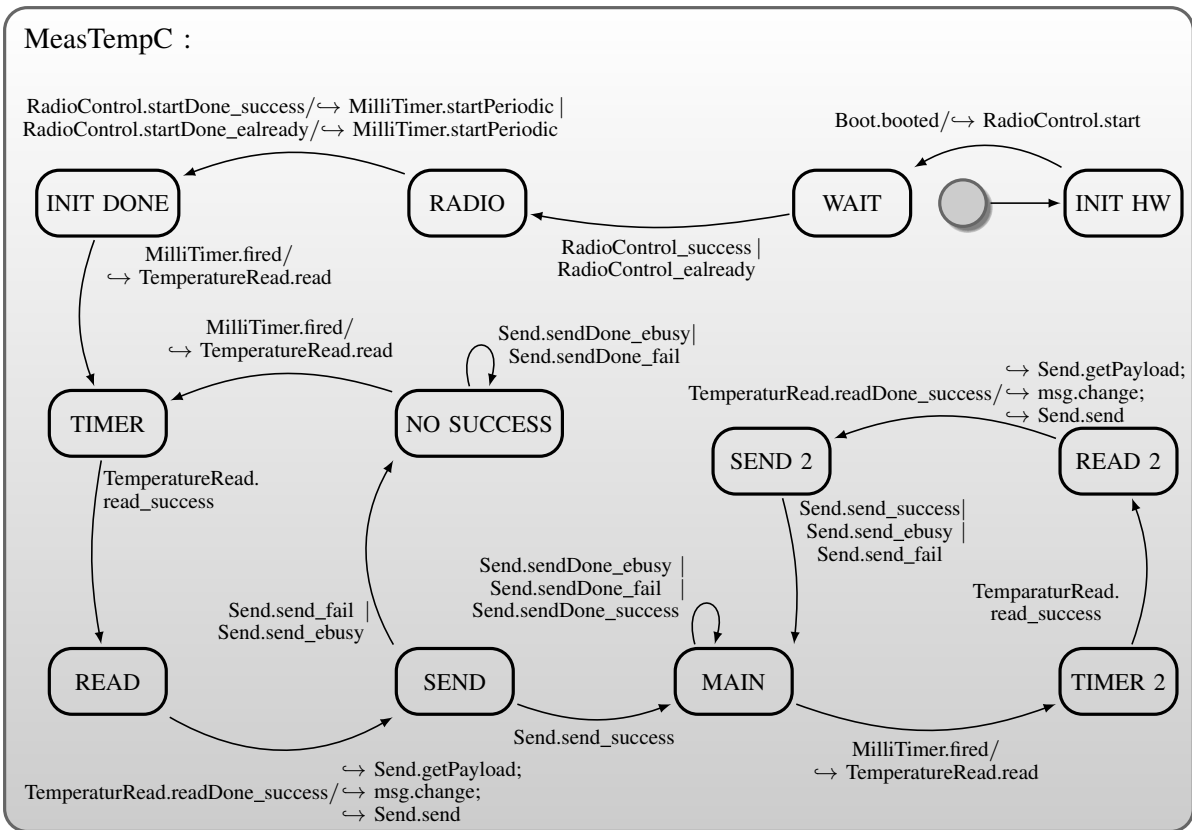


Fig. 3. MeasTempC Statechart

its region in state `OS_OWNED` (Figure 2), and `sendDone` has not been signalled yet; (b) After another temperature reading has been performed (i.e., the events `MilliTimerFired` and `TemperatureRead.read_success` have been consumed) the current state is `READ 2`. At this point, in transitioning to `SEND 2`, (c) the event triggered (`msg.change`) cannot be processed in state `OS_OWNED` in region `msg` (`msg.getPayload` could also not have been processed). This execution should not happen, as [5, §3.5.1] forbids it.

As hinted by the comments in Listing 2, the just described error can be avoided by keeping track in a boolean variable `packetOSOwned` of whether updates can be performed or not.

## V. RELATED WORK

Model-checking of nesC code has been the focus of several projects reported in the literature, covering a range of areas: radio-link modeling [6], communication protocols [7], and security properties [8], among others. Most of these works focus on the distributed scenario, where a correct implementation comprises several motes cooperating to achieve some goal. Properties of interest are the absence of

deadlock, that flooding messages reach all motes, and so on.

Checking the correctness of API usages was addressed first by Archer et al. [2] using a version of TinyOS extended with pre- and postconditions for runtime checking. Basu et al. [9] use BIP to model-check a motes network by composing components representing the user application (with annotations for model-checking), the operating system, and radio-links. Properties related to timing are verified (e.g., timer interrupts arriving while task processing is still going on).

More recently, static analysis techniques other than model-checking have been applied to nesC, e.g., McInnes [10] uses the process algebra CSP to model TinyOS scheduling and preemption mechanisms, which allows detecting state violations (e.g., trying to transmit a second sensor reading before transmission of the first has completed, which our approach also detects).

Bucur and Kwiatkowska [11] extend the SAT-based encoding of ANSI C performed by SATABS to model TinyOS system calls. Subsequently, concurrency errors due to unexpected interleavings can be detected, as well as a number of additional abnormal

conditions (network failure, interface use, sensor failure).

## VI. FUTURE WORK

The case study reported in this paper shows the feasibility and advantages associated to applying verification techniques early in the software development process. As future work, two next steps are (a) preparing behavioral contracts for all TinyOS interfaces; and (b) automating the verification of TinyOS user programs, by making such verification part of the normal build process.

One candidate engine for accomplishing (b) above is the SMUML [12] tooling for statechart verification. SMUML accepts a large subset of UML2 statecharts including an expressive action language, and uses two model-checkers as backends (SPIN and nuSMV). These tools automate the approach described in this paper, provided that statecharts have been recovered from nesC, a task requiring nesC software analysis.

In order to provide automated analyses for nesC, an additional software component is needed: we plan to customize a front-end for the nesC compiler. For this, a more general C-processing tool, CIL [13], is useful. The customizations will take care of parsing and reformulating nesC-specific syntax into plain C, after which CIL proper takes over.

## VII. CONCLUSIONS

The kind of bugs reported in the case study of this paper is not new. However, manually applying the coding idioms required to avoid those errors is taxing and unforgiving. Our approach improves productivity and quality in TinyOS development by warning about erroneous program logic at compile-time.

As new capabilities are added to TinyOS (e.g., threads) exploiting them to their fullest will require improved verification techniques. In retrospect, the difficulties in applying such techniques to nesC emphasize the need for designing languages and libraries from the start with verification in mind. We would like to see therefore more input from the

verification community addressing language design aspects for sensor networks, in addition to addressing existing programming models.

## REFERENCES

- [1] B. Meyer, "Applying Design by Contract," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.
- [2] W. Archer, P. Levis, and J. Regehr, "Interface contracts for TinyOS," in *IPSN '07: Proc. of the 6th Intl. Conf. on Information Processing in Sensor Networks*. New York, NY, USA: ACM, 2007, pp. 158–165.
- [3] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.
- [4] N. Walters, "Using Harel statecharts to model object-oriented behavior," *SIGSOFT Softw. Eng. Notes*, vol. 17, no. 4, pp. 28–31, 1992.
- [5] P. Levis and D. Gay, *TinyOS Programming*, 1st ed. Cambridge University Press, April 2009.
- [6] C. Appold, "Reliable model checking for WSNs," in *Proc. of the 8th GI/ITG KuVS Fachgespräch "Drahtlose Sensornetze"*. TU Hamburg-Harburg, August 2009, [http://www.ti5.tu-harburg.de/events/fgsn09/proceedings/fgsn\\_079.pdf](http://www.ti5.tu-harburg.de/events/fgsn09/proceedings/fgsn_079.pdf).
- [7] P. Völgyesi, M. Maróti, S. Dóra, E. Osses, and A. Lédeczi, "Software composition and verification for sensor networks," *Sci. Comput. Program.*, vol. 56, no. 1-2, pp. 191–210, 2005.
- [8] Y. Hanna, "SLEDE: lightweight verification of sensor network security protocol implementations," in *ESEC-FSE '07: Proc. of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. New York, NY, USA: ACM, 2007, pp. 591–594.
- [9] A. Basu, L. Mounier, M. Poulhiès, J. Poulou, and J. Sifakis, "Using BIP for modeling and verification of networked systems – A case study on TinyOS-based networks," in *NCA*. IEEE Computer Society, 2007, pp. 257–260.
- [10] A. I. McInnes, "Using CSP to model and analyze TinyOS applications," in *ECBS '09: Proc. of the 2009 16th Annual IEEE Intl. Conf. and Workshop on the Engineering of Computer Based Systems*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 79–88.
- [11] D. Bucur and M. Kwiatkowska, "Towards software verification for TinyOS applications," in *Proc. Workshop on Formal Approaches to Ubiquitous Systems (FAUST '09)*, September 2009, to appear.
- [12] J. Dubrovin and T. Junttila, "Symbolic model checking of hierarchical UML state machines," Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, Technical Report B23, December 2007.
- [13] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of C programs," in *Compiler Construction*, ser. LNCS, R. N. Horspool, Ed., vol. 2304. Springer, 2002, pp. 213–228.