AUTOMATIC FORMAL VERIFICATION FOR EPICS

Jonathan Jacky*, Stefani Banerian, University of Washington Medical Center, USA
Michael D. Ernst, Calvin Loncaric, Stuart Pernsteiner, Zachary Tatlock, Emina Torlak,
Paul G. Allen School of Computer Science & Engineering, Seattle, WA 98195, USA

Abstract

We built an EPICS-based radiation therapy machine control program and are using it to treat patients at our hospital. To help ensure safety, the control program uses a restricted subset of EPICS constructs and programming techniques, and we developed several new automated formal verification tools for this subset.

To check our control program, we built a *Symbolic Interpeter* that finds errors in EPICS database programs, using symbolic execution and satisfiability checking. It found serious errors in our control program that were missed by reviews and testing.

To check the EPICS runtime (EPICS Core) itself, we first developed a *Formal Semantics* for EPICS database programs, based on the EPICS Record Reference Manual (RRM) and expressed in the specification language of an automated theorem prover. We built a formally-verified *Trace Validator* and used it to check the EPICS runtime against our semantics by differential testing with millions of randomly generated programs. The testing process generally corroborated that the EPICS runtime conforms to its specification in the RRM, but it did find several omissions and ambiguities in the RRM that might mislead users. Our formal semantics for EPICS enables valuable future developments: a full proof of correctness for our EPICS program, verified analyses for arbitrary EPICS programs, and a *Verified Compiler* that could compile an EPICS database to a verified standalone program, while dispensing with much of the unverified EPICS toolchain and runtime.

INTRODUCTION

The Clinical Neutron Therapy System (CNTS) at the University of Washington Medical Center (UWMC) has been treating cancer patients with radiation therapy and making isotopes since 1984 [1]. The system includes a cyclotron and a treatment room with an isocentric gantry and leaf collimator operated under computer control. The system was built and installed by a vendor, but since then has been maintained and upgraded by UWMC staff. In 1999 we replaced the therapy portion of the vendor's original control system (a PDP11 programmed in FORTRAN) with new hardware and our own software (a 68040 in a VME crate running VxWorks programmed in C) [2]. In 2015 we replaced the therapy control hardware and software again, this time using the Experimental Physics and Industrial Control System (EPICS) running under Linux on an x86 processor [3].

EPICS has been used for over twenty-five years in many demanding applications [4–6]. Nevertheless, some experienced EPICS developers caution against using it for safety-critical controls:

> (EPICS) code is not rigorously audited to the standards … that would be needed (for medical applications). [7]
>
> EPICS should never be relied on for safety-critical operations [8]

Despite these warnings, we use EPICS at CNTS on some safety-critical signal paths essential for therapy. Our therapy control system uses relays, PLCs, and other non-EPICS components for safety-critical functions nearest the hardware, but we do use EPICS to process some of the data that is input to these components, and to process output that is collected for record keeping. In particular, we use EPICS to retrieve stored prescriptions from a database (comprising about 50 parameters per treatment field), load them into the control hardware, check conformance between the hardware and the prescription, store treatment records, and to restore and finish interrupted treatment sessions. Programs written in high-level languages running on general-purpose computers are best suited to handle these processing steps. For this, EPICS is no worse than the alternatives and offers some advantages. This judgment is based on many years of experience at our installation with EPICS and several alternatives.

We have always had a safe system design and a careful development process [2, 3, 9]. However, in view of warnings from experienced EPICS developers, the greater complexity of EPICS compared to our previous platforms, and the lack of any close precedents for our project, we felt an obligation to focus exceptional care and scrutiny on the EPICS components in our system.

We chose two complementary approaches. First, we limited ourselves to a restricted subset of EPICS components and programming styles that we expect to be feasible to understand, review, and analyze. Second, we developed specialized software tools for analyzing both our own code (an EPICS database, or program, written in our restricted style) and the EPICS runtime (the EPICS implementation, also known as EPICS Core). These remedies are complementary, since our restricted programming style limits the amount of EPICS our tools must model. For the latter, CNTS staff are collaborating with faculty and graduate students from the University of Washington School of Computer Science and Engineering (UW CSE) who are experts in the formal verification of software.

* jon@uw.edu. Thanks to the UWMC cyclotron engineering group: Dave Argento, Eric Dorman, Robert Emery, and Gregg Moffett.

ISBN 978-3-95450-193-9

We have developed and used two tools to help build reliable software using EPICS: a *Symbolic Interpreter* and a *Trace Validator*. The Symbolic Interpreter detects errors in EPICS programs (also known as EPICS databases). It has already found serious errors in our control program code that escaped detection by reviews and unit testing. The Trace Validator checks that the EPICS runtime adheres to our expectations by differential testing. We found some omissions and ambiguities in the documentation but no outright errors. In the course of building and using this tool, we performed a rigorous independent review of the documentation and code for the portions of the EPICS runtime that we use. We have thereby mitigated one of the objections made in [7]. In the future, we plan to create a *Verified Compiler* for generating, from an EPICS database, a standalone executable program that would not depend on the large, unverified EPICS runtime.

Some this work has already been reported in the computer science literature [10, 11]. This present report concentrates on the application of that work to our control program, including validation and workflow.

RESTRICTED EPICS CODING STYLE

Even before we contacted UW CSE, the CNTS staff resolved to use as few EPICS features and mechanisms as possible in order to make our program easier to understand and analyze. Specifically, we decided that the program should express a data flow from inputs to outputs, as simply as possible. The tools only work on EPICS application programs coded in this restricted style.

The entire therapy control program runs on a single soft IOC[1], the only application program on its computer. The IOC is self-contained; it does not need to communicate with any clients to achieve or maintain a safe state.

The entire IOC is expressed by EPICS database records (that is, the EPICS program), `StreamDevice` protocol files, and the IOC startup command file `st.cmd`. There is no custom device support nor any other custom code, not even subroutine records. We do not use the EPICS State Notation Language or the sequencer. The IOC itself does not use the EPICS Channel Access (CA) network protocol. A client database program uses CA to load prescribed settings into the IOC when a prescription is selected, and a client operator's console program uses CA to display PV values and transmit operator's commands, but the IOC itself does not require either of these to execute control laws or maintain safety. The database itself uses no CA links, only local database links (except in three instances, where we use a CA link to break a lockset for performance). All control flow in the database is "pushed" from input to output, using `SCAN PASSIVE`, `OUT PP`, and `FLNK` fields. We forbid run-time alteration of control or data flow within the EPICS database: we prohibit reassigning of fields such as input and output links or CALC expressions after IOC startup.

[1] An IOC is an EPICS Input-Output Controller: an executable built from an EPICS database program along with the EPICS runtime.

Our program uses only 19 record types: `acalcout`, `ai`, `ao`, `asyn`, `bi`, `bo`, `calc`, `calcout`, `dfanout`, `fanout`, `longin`, `longout`, `mbbo`, `scalcout`, `seq`, `stringin`, `stringout`, `subArray`, and `waveform`.

EPICS SYMBOLIC INTERPRETER

The *Symbolic Interpreter* finds errors in EPICS programs [12]. To use it, the application programmer codes an assertion that expresses what the program is intended to do, then runs the symbolic interpreter to see if the assertion fails. The symbolic interpreter checks the assertion for all possible input values. If the symbolic interpreter reports no assertion failure, then the assertion will be always be satisfied at run time.

Unlike unit testing, the programmer does not need to choose concrete values for the inputs. Another difference is that test coverage is usually incomplete, and a passing test is inconclusive—perhaps some other input values would have caused it to fail.

We consider the symbolic interpreter complementary to unit testing. We still do thorough unit testing, because it tests the actual IOC with EPICS Core as it will run in production, while the symbolic interpreter relies on a separate reimplementation of EPICS record processing logic.

Using the Symbolic Interpreter

The programmer provides three items to the symbolic interpreter: the program, an entry point into the program, and a property that the program's behavior should satisfy.

The program is specified by the `st.cmd` file that starts the IOC. The symbolic interpreter infers from it the program or database for the IOC, comprising all of its `.db`, `.substitutions`, and `.template` files.

The entry point is a record in the database; processing this record is the event whose consequences are analyzed. Typically we examine records where processing begins, such as those set to periodic scanning or those attached to physical devices.

The property expresses the intended behavior of the IOC in response to the given event. The property includes *preconditions*, which describe the values of pertinent EPICS process variables (PVs) before the event, and an *assertion*, the relation that should hold between PV values before and after the event, if the event occurs when the preconditions are all true.

For example, we used the symbolic interpreter to verify that when the measured gantry rotation angle (a therapy machine setting) differs from the prescribed gantry angle, subsequent processing will set an interlock that turns off the neutron beam and prevents it from turning on with the gantry in the wrong position. Figure 1 shows this property expressed in Racket, as it would be input to the symbolic interpreter.

A unit test could check this property for some (usually small) sample of prescribed and measured angles; the sym-

ISBN 978-3-95450-193-9

```
;; Read values from the hardware.
(define gantry-prescribed
  (get-value "Iso:GantryCouch:Gantry:Prescribed"))
(define gantry-actual
  (get-value "Iso:GantryCouch:Gantry:Actual"))
(define gantry-override
  (get-value "Iso:GantryCouch:Gantry:Override"))
(define session-mode
  (get-value "Iso:Session:Mode"))

;; Precondition:
;; The prescribed and actual (measured) angles differ.
(define precond
  (and
    (> (difference gantry-prescribed gantry-actual) 1)
    (= gantry-override 0)  ; Not in manual override
    (= session-mode 0)     ; Not in experiment mode
  ))

;; Run the record that reads the actual angle;
;; this also runs downstream records.
(process "Iso:GantryCouch:Gantry:Actual")

;; Assertion:
;; If precondition was true, then the beam is disabled.
(assert
  ;; In Racket, "=>" means implication
  (=> precond
      (= 0 (get-value "Iso:GantryCouch:Gantry:Calc"))))
```

Figure 1: Property (expressed in Racket) that expresses a safety requirement. The precondition states that the prescribed and actual (measured) angles differ. If so, then the beam must be disabled.

bolic interpreter checks all possible combinations of prescribed and measured values.

If the property is satisfied, the checker prints, "Everything is OK". If the property is violated, the checker prints a *counterexample* that demonstrates the violation.[2] The counterexample contains the values of the pertinent PVs in the initial state before the event begins and a log showing the chain of record processing steps, with the PV values for each, leading up to the violation. The counterexample is usually sufficient to identify the error that caused the failure, so it can be corrected.

The symbolic interpreter did find an error in the control program, and produced the counterexample shown in Figure 2. The actual angle differs from the prescribed angle, but the code erroneously indicates a match, setting the `Gantry:Calc.VAL` PV to 2 rather than the expected 0.

How the Symbolic Interpreter Works

To a programmer, the symbolic interpreter works like an exhaustive tester. Internally, it works symbolically: rather than storing a concrete value for each PV, it stores a set of constraints, such as "`VAL` is odd" or "`B` is positive if `A` is negative". It never runs the program explicitly, but analyzes every possible execution. The symbolic interpreter works in two stages, currently provided in two separate commands.

[2] The symbolic interpreter may also time out. This has never happened to us.

```
counterexample:
Iso:GantryCouch:Gantry:Actual.VAL = 48 [64-bit]
Iso:GantryCouch:Gantry:Prescribed.VAL = 312 [64-bit]
Iso:Session:Mode.VAL = 0 [16-bit]
Iso:GantryCouch:Gantry:Override.VAL = 0 [16-bit]

log:
start: Iso:GantryCouch:Gantry:Actual
  start: Iso:GantryCouch:Gantry:Calc
    Iso:GantryCouch:Gantry:Calc.A = 312
    Iso:GantryCouch:Gantry:Calc.B = 48
    Iso:GantryCouch:Gantry:Calc.C = 1
    Iso:GantryCouch:Gantry:Calc.D = 0
    Iso:GantryCouch:Gantry:Calc.VAL = 2
    ...
```

Figure 2: Counterexample generated by the symbolic interpreter that reveals an error in the gantry angle code. The symbolic interpreter log also shows the execution of downstream records (elided here for brevity).

In the *translation* stage, the interpreter reads the program (the database files) and generates a *symbolic IOC*: a simulator for the IOC expressed in the Racket language, where every PV is represented by the symbolic formula that calculates that PV value from all of the other PVs that might affect it. The formula is generated by accumulating all the CALC fields along all possible data flow paths to that PV, and might include logical and conditional operators as well as arithmetic operators.

The *checking* phase uses the symbolic IOC and the property. It combines the two into a single large formula such that any solution to the formula represents a violation of the property, and the absence of solutions indicates the property is satisfied. It invokes *Z3* [13, 14], an *SMT solver*, to solve the generated formula for PV values that violate the property. The checker is coded in *Rosette* [15, 16], a Racket package that simplifies the interface to Z3 and the task of generating formulas.

The symbolic interpreter only works for programs where processing of every event terminates, there are no loops, and there is no unbounded allocation of resources. These requirements are not ensured by programs in general. Termination is a common property of event processing in EPICS programs, because EPICS uses the PACT record field to prevent infinite loops while processing record chains. Our restricted programming style ensures that there are no loops or unbounded allocation of resources.

Validating the Symbolic Interpreter

If the symbolic interpreter is buggy, then the checker might print "Everything is OK" even though the EPICS program is defective. In addition to standard practices such as code review, we tested the symbolic interpreter to increase our confidence in it.

We seeded errors into EPICS programs, then we confirmed that the checker finds the expected counterexamples. We inspected the counterexample PVs and logs to confirm they show the expected IOC state and activities. We con-

ISBN 978-3-95450-193-9

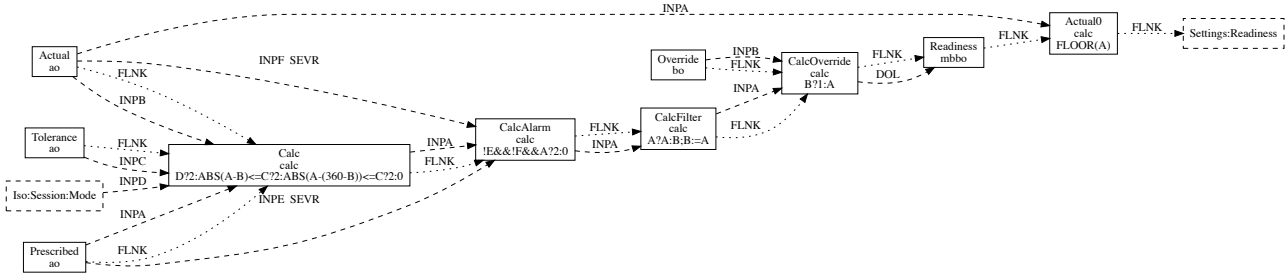

Figure 3: A small portion of our EPICS program that checks gantry angle. Each box represents an EPICS record. The entire program comprises over 1700 records. The "Calc" record in this portion incorrectly checks for similarity between the prescribed gantry angle ("A") and actual angle ("B"), causing the system to accept gantry angles that are opposite the prescribed value.

firmed that the counterexamples correspond to errors in real EPICS IOCs built from the same erroneous programs, and the counterexample logs resemble traces generated in the real IOC by EPICS after setting `TPRO` fields.

This provides confidence that the checker accurately models the real IOC, and we have detected a few errors in the checker this way. Primarily we use the counterexamples to develop and debug our properties.

Writing Properties for the Symbolic Interpreter

It can be challenging to write properties. Our first attempts to write a property often resulted in spurious counterexamples (i.e., normal behavior), due to missing preconditions. Examining the spurious counterexamples enabled us to correct the property and re-run the symbolic interpreter.

Sometimes the assertion is only intended to hold in particular modes or states, and the missing preconditions must precisely describe what mode or state of the IOC.

Another type of missing precondition restricts the values of a PV to those that can occur during real processing. Without an explicit precondition, the checker considers all possible values for the PV's data type and often produces counterexamples that rely on PVs taking on unrealistic or out-of-range values. Given a range constraint for a PV, the checker can ensure that it is preserved by certain events, but cannot infer such properties on its own.

Expected counterexamples can fail to appear when the preconditions are too strong so they exclude the triggering condition for the seeded error, or when there is a data flow path that satisfies the assertion while bypassing the seeded error.

Errors Found by the Symbolic Interpreter

The symbolic interpreter has found subtle errors that escaped detection during reviews, unit tests, and system tests. Here are two examples.

One error was quite serious: the property about the gantry angle in the example above was violated! The erroneous program (Figure 3) failed to interlock only when the measured angle was a reflection of the prescribed angle around zero degrees (for example, it would allow 270 degrees where 90 was prescribed), due to an error in the code to convert all angles (including negative angles) to the range 0–359.

The checker also identified a potential version skew problem. The control program was originally written to use the EPICS `calc` library version 2-8. In version 3-0, the `calc` maintainers made an incompatible change in the array indexing convention for `acalcout` records [17]. Later, the symbolic interpreter was upgraded (at UW CSE) to use version 3-2 of the `calc` library, while the control program itself (at UWMC) continued to use the old 2-8 version. The checker accurately models version 3-2 of the library, so it was able to identify a situation in which the new array indexing conventions in the library, when called by the old, unrevised version of the control program, would allow the beam to turn on even when some machine settings differ from the prescribed values. Thus forewarned, the control program maintainers at UWMC made the necessary changes in the control program when they installed version 3-2 of the `calc` library.

The checker also identified a potential performance problem. A counterexample log revealed that a large number of records were being processed unnecessarily, due to redundant `FLNK` fields. The problem was confirmed in the actual IOC after setting `TPRO` fields. The redundant `FLNK` were removed, and the IOC functioned correctly while processing many fewer records.

FORMAL SEMANTICS

We have created a *formal semantics* for our subset of EPICS. The formal semantics precisely describes the set of possible behaviors for any given IOC. Our formal semantics is derived from the informal English specification found in the EPICS Record Reference Manual (RRM) [18] and has been validated through millions of differential tests against the EPICS runtime. The creation and validation of this formal semantics identified a number of omissions and ambiguities in the EPICS specification, which might lead to unexpected behavior for users.

To produce our formal semantics, we read the RRM entries for the record types we use, focusing particular attention

ISBN 978-3-95450-193-9

on the *Record Processing* sections. We then re-implemented those definitions in Coq [19], a mechanized proof assistant.

Coding the Formal Semantics

We decomposed all the activities described in the RRM into combinations of about thirty primitive operations that each perform a small change to the state of the IOC (its PV values), while possibly consuming inputs or producing outputs as well. Each record's process routine can be compiled into a sequence of these operations. Each primitive operation is individually very simple, and many are needed to process even a single record. For example, one primitive operation is to copy the value of one PV into another PV. Another is to load a constant into a PV.

The semantics are single-threaded. Operations are *atomic*: only one operation can execute at a time, and must complete before another operation can begin. This has the effect of modeling EPICS *lock sets*.

There are some sequencing constraints; certain sequences of operations can only execute in particular orders. We express these by writing *preconditions*, assertions that describe the states where a given operation can occur.

In addition to the operations that an IOC may perform on its own PVs, our semantics capture things that might happen in the outside environment. For instance, a new value might become available on a hardware device or a timed callback might fire.

The semantics admit nondeterminism. The results of some operations, such as reading from a hardware device, are not fully determined by the current state of the IOC, so the semantics allows multiple behaviors. For example, because the semantics does not require a particular outcome for a hardware read, the space of allowed behaviors will include one where the read produces 2.0, one where it produces 3.0, and so on for each possible reading of the hardware device.

VALIDATING THE FORMAL SEMANTICS

The EPICS runtime is correct if it behaves as specified in the RRM. Establishing this property would be valuable to the EPICS community.

Our formal semantics is correct if it accurately models the EPICS runtime. Any discrepancies would compromise its usefulness in identifying bugs in EPICS programs. Our formal semantics is based on our reading of the RRM.

Ideally, the RRM, our formal semantics, and the EPICS runtime would be consistent. We have not proved these correspondences, but we have used testing to identify discrepancies between the EPICS runtime and our formal semantics. Our testing identified a number of errors in early versions of our semantics, where we had incorrectly interpreted the RRM; we have since corrected these errors. More seriously, our testing identified several omissions and ambiguities in the RRM.

We built a *Trace Validator* that compares one IOC execution to our semantics. The trace validator takes as input a trace of events, such as record processing steps and changes to PVs, captured from an instrumented IOC, and checks that the given trace is consistent with the semantics. We ran the trace validator on traces from millions of randomly-generated programs to ensure that our semantics accurately describes the behaviors of real EPICS programs. This process of comparing two independent implementations is called "differential testing".

We performed this testing process because we were concerned about the complexity of the EPICS framework, toolchain, and runtime. This might lead either the EPICS maintainers, or EPICS programmers like us, to make errors. In particular, EPICS seems more complicated and harder to understand than previous platforms used by CNTS. We were also mindful of the warning that the EPICS "code is not rigorously audited" [7]. Validating our formal semantics constitutes an audit that the behavior of the EPICS database processing engine conforms to our understanding of how it is supposed to work, based on our reading of the RRM.

Coding the Trace Validator

The trace validator checks that each state transition observed in a trace matches an operation defined in our formal semantics.

We coded the trace validator in Coq. This makes it possible to verify the validator—i.e. use Coq to write and check a proof of correctness of the trace validator. Specifically, we proved in Coq that if the validator accepts a trace, then that trace is actually possible under our semantics. Put another way: if a trace is impossible under our semantics, then the validator will reject it.

Differential Testing

We randomly generated over 20 million small test cases. Each test case is a database of five records, where the trace is started by an EPICS IOC shell `dbtr` command. We used the trace validator to empirically check that our EPICS semantics is consistent with behavior of the EPICS runtime.

No test case caused the EPICS runtime to crash, but we discovered dozens of discrepancies. Most discrepancies were due to our misreading of the RRM and EPICS runtime code; we revised our semantics. Other discrepancies revealed ambiguities and omissions in the documentation. Here are some examples:

- The RRM does not document the default value for `OMSL` on `dfanout` records and others. We assumed the default was `closed_loop`, but the EPICS runtime sets it to `supervisory`.
- `calcout` records with `OOPT = On Change` consider $inf \neq inf$ for the purposes of determining when to execute their output link, contrary to IEEE 754 [20].
- `seq` uses callbacks even if all delays are zero. The callbacks are chained, not scheduled simultaneously, which allows them to interleave with other records processing.

ISBN 978-3-95450-193-9

- Records with several input links will store the value read from the link into a local field before proceeding to the next link. A `calc` record, for instance, will write to its `A` field before fetching the value pointed to by `INPB`. This is observable if later input links refer to earlier fields or if an input link processes a record that reads from an earlier field.

There might be other omissions in the RRM, which we did not notice because we made an assumption that is the same as the EPICS runtime behavior. An advantage of our formal semantics is that it makes every decision explicit, so all assumptions are clear to anyone who reads the formal semantics.

None of the discrepancies above affected the therapy control program. They yield differences in behavior only in configurations that do not occur in our code. The testing increases our confidence that the records we use, when used in our restricted style, behave as described in the EPICS RRM and as implemented in the EPICS runtime.

FUTURE WORK: VERIFIED EPICS TOOLCHAIN

We plan to use our formal semantics to produce verified tools for EPICS programs. We have already created two proof-of-concept verified tools: a range analysis and an interpreter. An ambitious goal is a *verified compiler*, that would compile an EPICS database to a verified standalone program that could replace an EPICS IOC, while dispensing with much of the unverified EPICS toolchain and runtime.

Verified Analysis

Our semantics allows us to verify that analyses we write for EPICS programs are correct. For example, the Symbolic Interpreter is a powerful but unverified analysis. Future work is to verify it with respect to our semantics.

We have already built one verified analysis, which finds PVs that can only take on a finite set of values (e.g. zero or one). Its proof of correctness states that if the analysis reports a given PV can only be zero or one, then no matter what happens to the IOC, that guarantee will hold. We hope to use this analysis to help check some of our preconditions for symbolic interpreter properties.

Verified Compiler

We have already built a *verified interpreter*, which runs EPICS programs much like EPICS would, but without device support or channel access. We defined an opcode for each operation in our formal semantics, then coded (in Coq) an interpreter that processes each opcode by assigning PVs to update the IOC state and (where required) by consuming inputs and producing outputs. We then used the Coq theorem prover to check that the behavior of this interpreter is consistent with the assertions in the formal semantics. The proof confirms the interpreter does not make any updates that violate the assertions in our semantics.

We hope to use a similar approach to produce a re-implementation of EPICS with sufficiently good performance to build IOCs that could be used in production. This would provide an alternative to the complicated EPICS toolchain and much of the large, unverified EPICS runtime.

We plan to use the same formal semantics and opcodes, but then perform a verified transformation to a program in C. We would then generate machine code via a C compiler that was proved correct, such as CompCert [21].

CONCLUSIONS

It is possible to use EPICS with confidence in safety-critical applications, by using the programming style and the tools described here.

We have demonstrated:

1. successful use of restricted EPICS in a safety critical application,
2. a tool that finds errors in EPICS databases using exhaustive analysis,
3. testing of the EPICS runtime against the RRM finds ambiguities and omissions, but no crashes or outright errors.

Those interested in obtaining the toolset can contact the first author at `jon@uw.edu`.

REFERENCES

[1] R. Risler, S. Banerian, J.G. Douglas, R.C. Emery, I. Kalet, G.E. Laramore, and D. Reid. 25 years of continuous operation of the Seattle clinical cyclotron facility. In Youjin Yuan, Lina Sheng, and Lijun Mao, editors, *Proceedings of the Nineteenth International Conference on Cyclotrons and Their Applications*, pages 68–70, 2010.

[2] Jonathan Jacky, Ruedi Risler, David Reid, Robert Emery, Jonathan Unger, and Michael Patrick. A control system for a radiation therapy machine. Technical Report 2001-05-01, Department of Radiation Oncology, University of Washington, Box 356043, Seattle, Washington 98195-6043, USA, May 2001. `http://staff.washington.edu/jon/cnts/cnts-report.html`.

[3] Jonathan Jacky. EPICS-based control system for a radiation therapy machine. In Christopher Marshall, John Fisher, and Volker R. W. Schaa, editors, *Proceedings, 14th International Conference on Accelerator & Large Experimental Physics Control Systems (ICALEPCS 2013)*, pages 922–925, 2014. `http://accelconf.web.cern.ch/AccelConf/ICALEPCS2013/papers/tucoca05.pdf`.

[4] L. R. Dalesio, M. R. Kraimer, and A. J. Kozubal. EPICS architecture. In C. O. Pak, S. Kurokawa, and T. Katoh, editors, *Proceedings of the International Conference on Accelerator and Large Experimental Physics Control Systems*, pages 278–282, 1991. ICALEPCS, KEK, Tsukuba, Japan.

[5] Matthias Clausen and Leo Dalesio. EPICS - Experimental Physics and Industrial Control System. *Beam Dynamics Newsletter*, (47):56–66, Dec 2008. `http://www-bd.fnal.gov/icfabd/Newsletter47.pdf`.

ISBN 978-3-95450-193-9

[6] http://www.aps.anl.gov/epics/.

[7] Andrew Johnson. Re: Is there anyone using EPICS for a FDA approved device? Tech-talk mailing list. Tue, 19 Aug 2008 10:56:21 +0200, http://www.aps.anl.gov/epics/tech-talk/2008/msg00797.php.

[8] Andrew Johnson. Re: Doubt regarding standards followed by EPICS. Tech-talk mailing list. Mon, 10 Sep 2012 16:52:29 -0500, http://www.aps.anl.gov/epics/tech-talk/2012/msg01836.php.

[9] Jonathan Jacky. Formal safety analysis of the control program for a radiation therapy machine. In Wolfgang Schlegel and Thomas Bortfeld, editors, *The Use of Computers in Radiation Therapy: XIIIth International Conference*, pages 68–70, Berlin, Heidelberg, New York, 2000. Springer. http://staff.washington.edu/jon/cnts/iccr.html.

[10] Michael D. Ernst, Dan Grossman, Jon Jacky, Calvin Loncaric, Stuart Pernsteiner, Zachary Tatlock, Emina Torlak, and Xi Wang. Toward a dependability case language and workflow for a radiation therapy system. In *SNAPL 2015: the Inaugural Summit on Advances in Programming Languages*, pages 103–112, Asilomar, CA, USA, May 2015.

[11] Stuart Pernsteiner, Calvin Loncaric, Emina Torlak, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Jonathan Jacky. Investigating safety of a radiotherapy machine using system models with pluggable checkers. In *CAV 2016: 28th International Conference on Computer Aided Verification*, pages 23–41, Toronto, Canada, July 2016.

[12] Calvin Loncaric. EPICS symbolic interpreter manual. University of Washington Department of Computer Science and Engineering.

[13] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] https://github.com/Z3Prover/z3.

[15] Emina Torlak and Rastislav Bodik. Growing solver-aided languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 135–152, New York, NY, USA, 2013. ACM.

[16] https://emina.github.io/rosette/.

[17] https://www3.aps.anl.gov/bcda/synApps/calc/calcReleaseNotes.html.

[18] http://wiki-ext.aps.anl.gov/epics/index.php/RRM_3-14.

[19] https://coq.inria.fr/.

[20] Institute of Electrical and Electronics Engineers. IEEE standard for binary floating-point arithmetic. 345 East 47th Street, New York, NY 10017, August 12, 1985. IEEE Standard 754-1985.

[21] https://compcert.inria.fr/.

ISBN 978-3-95450-193-9