A Formal Foundation for Failure Avoidance and Diagnosis

Terence Kelly, Yin Wang, Stephane Lafortune, Matt Welsh

HP Laboratories
HPL-2009-203

Keyword(s):
fault tolerance, failure avoidance, concurrent programming, sensor networks, discrete control theory

Abstract:

This paper argues that Discrete Control Theory (DCT) provides a useful formal foundation for failure avoidance and diagnosis in a wide variety of computing systems. Our experience applying DCT to several difficult systems problems during the past three years convinces us that this powerful, general, mature, and rigorous body of theory belongs in the standard dependability toolbox. It is particularly valuable in new contexts thrust upon us by recent technology trends, including sensor networks and the multicore revolution.

External Posting Date: August 21, 2009 [Fulltext] Approved for External Publication
Internal Posting Date: August 21, 2009 [Fulltext]

A Formal Foundation for Failure Avoidance and Diagnosis

Terence Kelly [1] Yin Wang [1,2] Stéphane Lafortune [2] Matt Welsh [3]

[1]*Exascale Computing Lab, HP Labs*

[2]*Discrete Dynamics & Control Lab, U. Michigan EECS*

[3]*Sensor Networks Lab, Harvard CS*

1 A New Tool for Changing Times

This paper argues that *Discrete Control Theory* (DCT) provides a useful formal foundation for failure avoidance and diagnosis in a wide variety of computing systems. Our experience applying DCT to several difficult systems problems during the past three years convinces us that this powerful, general, mature, and rigorous body of theory belongs in the standard dependability toolbox. It is particularly valuable in new contexts thrust upon us by recent technology trends, including sensor networks and the multicore revolution.

So why have you (probably) never heard of it? DCT is a relatively recent development, dating back to seminal work in the late 1980s [13]. The DCT research community is relatively small, publishes in control theory journals, and gravitates toward manufacturing and industrial control applications. Computer scientists and DCT specialists seldom seek each other out, even when they work in the same buildings. When the two tribes meet they speak different dialects, even when discussing the same problem. Our own work began as an accidental collision of chocolate with peanut butter: One of us (Wang), an intern on a project with no specific DCT mandate, repeatedly insisted to his mentor (Kelly), "there's a better way to do it!"

At a high level, DCT resembles classical control: Starting with a model of the "plant" to be controlled and a specification of behaviors to be avoided, DCT automatically synthesizes a controller such that the closed-loop system (plant + controller) cannot misbehave. Whereas classical control considers continuous state spaces and differential-equation dynamics, DCT addresses discrete state spaces and event-driven dynamics; it is therefore well suited to enforcing *qualitative* control objectives such as failure avoidance. DCT also includes *diagnostic* techniques that infer unobservable events from indirect evidence, and DCT can explicitly accommodate failures in the runtime controller's sensors and actuators. Finally, core DCT techniques admit *distributed* generalizations that allow federations of local modules to achieve the same objectives as a centralized controller/diagnoser with minimal communication.

DCT has many attractions for CS dependability researchers. It is not an abstruse technology accessible only to an elite priesthood. Its foundations involve formal languages, automata, and other elements familiar to CS undergrads; useful control synthesizer implementations can be compact, elegant, and intelligible to newcomers. Like parser generators, DCT operates upon declarative specifications that raise the level of abstraction and can dramatically simplify what had been the hard part of a problem. Like compiler optimizations, DCT can admit friendly interfaces that expose benefits to users without imposing burdens on them. By automatically compiling high-level policies into low-level imperatives, DCT can improve dependability by working across system layers while preserving layer boundaries. DCT shifts most of the computational burden of online diagnosis and control to offline pre-computations, thereby dramatically reducing the runtime overhead of analysis and decision-making.

We believe that the time has come for the systems dependability community to include DCT in its toolkit. Two seemingly unrelated technology trends, the rise of multicore processors and of sensor networks, strengthen our conviction because both contexts are prone to failures that DCT can alleviate. Multicore hardware requires parallel software for performance, and thus imposes the notorious pitfalls of concurrent programming upon vast numbers of ill-prepared developers. Sensor networks combine two of the most formidable programming challenges, distributed computing and severe resource constraints. We will demonstrate that DCT can solve thorny problems in the former context and holds great promise for the latter.

We begin by reviewing the concepts and capabilities of DCT in Section 2. Section 3 then describes our experiences applying DCT to dynamic failure avoidance in two very different concurrent programming environments. We consider DCT-based diagnosis in sensor networks in Section 4 and conclude in Section 5.

2 Discrete Control Theory

DCT has matured rapidly in the two decades since its inception. Comprehensive graduate-level texts summarize the most important results [1]. This section briefly reviews two categories of DCT techniques most relevant to systems dependability: *control* methods that can avoid

failures, and *diagnostic* methods that can infer failures (and other events of interest) from indirect evidence.

DCT control methods require two inputs: a system model and a behavioral specification. The two most popular DCT modeling formalisms are finite automata and Petri nets [11]. The former are fully general in terms of both their expressiveness and the control objectives that DCT can achieve, but can be excessively large for systems of interest. Petri nets can be more structured and therefore more compact than finite automata, but efficient DCT algorithms that operate directly upon Petri nets without explicitly enumerating their state spaces achieve less general control objectives. Models may designate state transitions as controllable or uncontrollable, and also as observable or unobservable at runtime. Behavioral specifications for finite automata models simply consist of states to be avoided. Specifications for Petri net models are typically linear-arithmetic constraints on their markings (states).

Modeling and behavioral specification need not be a burden on users. In our concurrent programming work, for example, we extract models automatically from program source code. In cases where automatic modeling is not possible, models simply formalize the human designer's understanding of system dynamics. Behavioral specifications can often be expressed as straightforward predicates on states. For the deadlock avoidance problems in our research, users need not supply an explicit specification; deadlock states are identified automatically.

DCT control synthesis outputs *augmented* versions of the system model inputs. For automata models, the augmented output is a labeling of *unsafe* states in the original model. To enter such a state is to tempt fate, because a worst-case sequence of uncontrollable state transitions could lead either to failure or to a state from which successful termination is unreachable. Control logic boils down to disabling controllable transitions into unsafe states. For Petri net models, the output is a larger Petri net to which control logic elements have been added. The dynamics of the new Petri net are such that states violating the behavioral specification are no longer reachable. Control synthesis fails with an error message if is impossible to control the given system to enforce the given behavioral specification; this can occur due to uncontrollable or unobservable state transitions. Implementations of DCT control logic synthesis can be remarkably simple and elegant. For automata models, control synthesis can involve repeated application of graph search algorithms; for Petri net models, control synthesis can involve elementary linear algebra.

Online control enforces the control policy computed during control synthesis: For automata models, online control simply refrains from entering unsafe states of the augmented automaton. For Petri net models, control constrains execution to conform to the augmented model generated by control synthesis. In both cases, the standard observation-action paradigm of feedback control is at work. DCT control synthesis for both automata and Petri net models can achieve the important property of *maximal permissiveness*, which means that runtime control intervenes only when provably necessary to prevent failure in a worst-case future execution.

Diagnostic techniques in DCT allow the runtime inference of unobservable events from indirect evidence. These techniques can be used to identify failures or any other phenomena of interest that cannot be directly observed but that have observable consequences. DCT can explicitly accommodate the possibility of sensor and/or actuator failure by including these possibilities in the system model; transitions to sensor/actuator failure states would typically be modeled as both uncontrollable and unobservable events. Diagnostic techniques can infer such failures using properly synthesized diagnoser automata or Petri nets, and DCT control synthesis can generate control logic that continues to ensure correct execution even in their presence.

So far we have discussed centralized DCT techniques, which assume that runtime control logic is unified in the sense that it observes all observable transitions and controls all controllable transitions. Distributed generalizations of core DCT techniques address situations where control and/or diagnostic logic must be distributed across modules with access to different subsets of observable and controllable transitions. The modules may communicate with one another, and could trivially emulate a centralized solution. DCT, however, allows us to minimize inter-module communication subject to the constraint that the ensemble of local modules must collectively achieve the stated control or diagnostic goals as effectively as a centralized controller/diagnoser would.

In summary, DCT techniques have many attractive properties for systems dependability problems. Behavioral specifications are declarative, allowing the system designer to say *what* she wants rather than *how* to get it. Computationally burdensome control logic synthesis occurs offline, and online control logic is lean and fast. DCT control logic synthesizers can be remarkably simple and elegant. The final closed-loop system obtained by applying DCT to an uncontrolled system is guaranteed to be correct by construction; no separate verification phase is needed.

3 Concurrent Programs: Avoiding Failure

Concurrency bugs have killed people [9], and they threaten widespread mayhem as multicore hardware compels ill-prepared developers to parallelize an ever

wider range of software in pursuit of performance [16]. We have applied DCT to dynamic failure avoidance in two very different concurrent programming environments. Our experiences highlight both the versatility of DCT techniques and tradeoffs between their breadth of applicability and benefits. This section summarizes our work in this area; details are available in [17–20].

IT Automation *Workflow* programming languages—restricted high-level scripting languages that emphasize control flow rather than data manipulation and that support concurrency across hosts in data centers—are increasingly popular for IT automation. Because they are used for business-critical tasks, these workflows must be dependable but must also accomplish their goals quickly. The need for speed implies exploiting concurrency where possible, but the dependability requirement recommends caution. For example, concurrency bugs in disaster-recovery workflows can exacerbate the crises that these workflows were intended to solve.

Fortunately, the restricted nature of workflow programming languages facilitates far more powerful static analyses than are possible for general-purpose programming languages. Static analysis, however, merely identifies defects; repair remains manual, costly, error-prone, and time-consuming. Manual maintenance programming is also required even for *correct* workflows when requirements change, which occurs frequently in modern data centers.

Our solution applies DCT to address both defects and maintenance. We automatically extract a model of a workflow from source code, then automatically synthesize a controller that dynamically avoids both standard pre-defined failures (e.g., deadlocks) and also *arbitrary user-specified undesirable execution states*. We were able to achieve this level of generality by exploiting the restricted nature of workflow programming languages: We model workflows using finite automata that explicitly represent the entire reachable state space of workflow execution. Remarkably, the state spaces of real workflows are precisely in the "sweet spot" for our approach: far too large for human programmers to reason about, but small enough for automata-based DCT control synthesis.

Our approach effectively "fixes" defective workflows by dynamically avoiding deadlocks and user-specified failures. Programmers can delegate to DCT responsibility for avoiding corner-case concurrency bugs, and concentrate on writing natural workflows instead of perfect ones. Finally, if changed requirements are expressed as DCT specifications, an automatically synthesized controller can guarantee compliance, eliminating the need for maintenance programming.

Extensive tests on real workflows bundled with a commercial workflow product demonstrate that our approach scales adequately in this domain. However a different approach is needed for general-purpose programs, whose execution state spaces are too large for finite-automata models.

Multithreaded Programming We next applied DCT to dynamic deadlock avoidance in C/Pthreads programs. As with workflows, we automatically extract a program model from source code. However we model C programs with Petri nets, which are more structured and therefore far more compact than finite automata. Our modeling approach establishes a correspondence between deadlock in a C program and structural features in the corresponding Petri net model; static analysis can identify these features. Our modeling and analyses are conservative: they never overlook potential deadlocks but sometimes detect them where none in fact exist. Succinct and convenient programmer-supplied function annotations improve the accuracy of our modeling, greatly reducing the number of "false positives." Remaining spurious deadlocks can reduce performance by causing superfluous control logic to be generated, but cannot cause safety or correctness violations.

During offline control logic synthesis, we instruct a DCT control synthesis algorithm to dynamically avoid deadlocks in the model. The algorithm's output is a Petri net augmented with a small number of new features that alter its dynamics in such a way as to guarantee deadlock-free execution. Finally, we instrument the original program with control logic that constrains it to behave like the augmented model. At runtime this control logic dynamically avoids circular-mutex-wait deadlocks by postponing lock acquisitions whenever necessary to avert deadlock in a worst-case future execution scenario. Deadlocks involving other primitives (e.g., condition variables) are handled analogously. Our approach provably guarantees the elimination of all deadlocks in the original program without introducing new deadlocks or livelocks and without adding, removing, or altering functionality.

Our approach can reduce performance, but extensive experiments on benchmark programs and on open source software including Apache, OpenLDAP, and BIND show that our prototype implementation successfully avoids both injected and naturally occurring deadlocks while imposing very modest performance overheads—typically negligible under realistic conditions and never more than roughly 18% under pessimistic conditions. Comparisons between lock-based and transactional-memory implementations of a performance benchmark show that our approach outperforms a commercial TM system by several measures and by large margins.

Several crucial DCT advantages contribute to the low overhead of our approach. Our control logic synthesis procedure ensures *maximally permissive control*, which

in our domain implies that, e.g., lock acquisitions are postponed only when necessary to ensure that deadlocks cannot subsequently occur. This property translates directly into maximal runtime concurrency. Just as importantly, DCT performs nearly all of the computation required to achieve deadlock avoidance during *offline* static analysis and control logic synthesis. In essence, DCT allows us to perform a deep whole-program static analysis and compactly encode prepackaged context-sensitive decisions, allowing runtime control to quickly adjudicate lock acquisition requests based on current program state and worst-case future execution possibilities. The runtime checks required to ensure deadlock avoidance for real software require *constant* time. A final performance advantage is that the control logic we embed in a program is local, fine-grained, and highly concurrent. There is no "big global lock" protecting controller state nor any other global performance bottleneck.

In contrast to static analysis alone [5], our approach automatically fixes potential deadlocks in addition to identifying them. In contrast to recent proposals to *dynamically* detect then dynamically avoid *re-occurrences of deadlocks* [6, 12], our approach statically eliminates all deadlocks in a single stroke, ensuring that they never occur in production—not even once. The lightweight, local, highly concurrent runtime checks of our approach furthermore contrast with the global serial bottlenecks and expensive safety checks required by online deadlock detection.

4 Sensor Networks: Distributed Diagnosis

DCT provides a compelling methodology for diagnosing failures in distributed systems with expensive internode communication and in situations where the runtime computational cost of control actions must be minimized [4, 15]. Wireless sensor networks are a good example of such systems. In addition to using DCT as a means for decentralized failure diagnosis in the network, we believe that DCT can provide an effective network-wide "macroprogramming" environment that allows one to describe the network's behavior at a global level.

Sensor networks are characterized by potentially many nodes operating in a very resource-constrained setting, in which computational power, memory availability, and communication bandwidth are extremely limited. In low-duty-cycle settings, it is infeasible to rely on centralized monitoring, making it especially difficult to diagnose failures. Common failure modes include software hangs and reboots, deadlocks, memory leaks, loss of radio connectivity, and energy exhaustion. Many failure modes lead to Byzantine behaviors, causing nodes to flood the network with bogus packets or corrupting the state of a routing protocol [2].

DCT is well-suited to addressing this challenge by providing a rigorous approach to characterizing the state of individual sensor nodes and the network as a whole, addressing the limited observability of the state of the system. DCT can also capture distributed cases in which multiple diagnosers with different views of the overall system state track event transitions and make independent diagnostic decisions. Upon detection of an undesirable state, the DCT controller can perform corrective actions such as re-initializing the routing protocol or rebooting a node. Hence, DCT can be used for both decentralized failure detection *and* mitigation.

Current approaches to sensor network failure detection tend to rely on centralized observation of the network state. Systems such as Sympathy [14] and LiveNet [2] collect periodic health reports from each node or detailed traces of radio communication activity, and subject this raw information to multiple levels of analysis at a central base station to determine the cause of failure or performance issues. For example, in Sympathy, each sensor node reports metrics including neighbor tables, packet transmission and reception counts, and uptimes back to the centralized sink node. The sink uses a taxonomy of common failure modes including node crash, reboot, disconnection, and routing path failures to determine the root cause of a failure. Apart from the high overheads involved in collecting these metrics centrally (especially in cases where the network operates at a very low duty cycle), this approach is incapable of detecting faults "deep in the network" not observable at the sink node.

Building on the work in [4, 15], DCT could synthesize an automatic failure diagnoser that runs efficiently on each sensor node, driven by local observations of the sensor node's state as well as information from its one-hop neighbors, such as explicit state messages or snooped traffic. The DCT model would consist of states representing both normal and abnormal behavior. Examples of "normal" states include node liveness (such as whether packets have been received from the node), whether a node appears to be time synchronized with its neighbors (based on comparison of each node's idea of the global clock), and whether the current routing path appears to be loop-free.

If the DCT diagnoser evolves to an "abnormal" state, the node can transmit a message indicating that it has detected a failure locally or within its neighborhood, which may be supported or refuted by observations at nearby nodes. Upon confirmation of the failure more information can be delivered to the sink. Of course, DCT can also be used to control the evolution of each node's state, although in the case of sensor networks the most effective intervention may be to reboot a node, given the assumption of soft state and spatial redundancy made by most sensor network applications. DCT can enable a prin-

cipled failure-detection framework for sensor networks that factors out important diagnosis functions common to many applications.

Beyond failure detection within the sensor network itself, we envision that DCT could provide an effective framework for *macroprogramming* complex, network-wide behaviors. Typical sensor network applications consist of event-driven state machines that evolve through the collection of local sensor data and the reception of radio messages, which can be readily captured with an automaton or Petri net representation. This approach could allow us to describe the state of the network as a whole, rather than that of individual sensor nodes. DCT can then be used to synthesize the *node-level* program (control logic) directly from this high-level specification. This approach represents a radically different way of thinking about sensor network programming and might be used to produce more robust and efficient solutions automatically.

As an example, consider a sensor network tasked to detect structural faults in a building, bridge, or other structure [3, 7]. In these systems, each node performs periodic sampling of an accelerometer, producing a vibration signature on each sampling interval. Combining these vibration signatures from nearby nodes permits segments of the network to determine if the structural response conforms to an expected envelope, or whether the structure is experiencing a fault (crack, deformation, etc.) or sudden shock (such as an earthquake).

In this case, the high-level logic for combining multiple sensor observations into a single event report is relatively straightforward, and can be described by a simple state machine. However, *implementing* this logic on sensor nodes is currently fairly involved, requiring the use of asynchronous data sampling interfaces, timers, low-level radio communication interfaces, memory management, and a multitude of knobs to optimize power consumption [8]. Using DCT, the high-level operation of the network can be captured, in this case for *detecting failures in the environment* observed by the sensor network, rather than within the sensor network itself. The resulting state-transition diagram can be compiled down to an efficient node-level program, for example, using an intermediate language such as Flask [10]. For a certain class of sensor network applications we find the DCT approach to program synthesis compelling and expect it will yield new insights into automatic generation of node-local programs from global specifications.

5 Conclusion

We have argued that Discrete Control Theory holds great promise for a wide variety of systems dependability problems. Extensive experience applying DCT to concurrent programming and our ongoing work applying DCT to sensor networks convinces us that DCT belongs in the toolbox of every systems dependability researcher. Our experience has taught us that the most vexing challenges confronting the dependability community today—complexity, opacity, concurrency, resource constraints, and large-scale decentralization—*play to the strengths* of Discrete Control Theory.

References

[1] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer, second edition, 2007.

[2] B. Chen, G. Peterson, G. Mainland, and M. Welsh. Livenet: Using passive monitoring to reconstruct sensor network dynamics. In *Conf. on Dist'd Comp. in Sensor Sys. (DCOSS)*, June 2008.

[3] K. Chintalapudi et al. Monitoring civil structures with a wireless sensor network. In *IEEE Internet Computing*, March/April 2006.

[4] R. Debouk, S. Lafortune, and D. Teneketzis. Coordinated decentralized protocols for failure diagnosis of discrete-event systems. *Discrete Event Dynamic Systems: Theory and Applications*, 10(1/2), 2000.

[5] D. Engler and K. Ashcraft. RacerX : effective, static detection of race conditions and deadlocks. In *SOSP*, 2003.

[6] H. Jula and G. Candea. A scalable, sound, eventually-complete algorithm for deadlock immunity. In *Wkshp on Runtime Verification*, Mar. 2008.

[7] S. Kim et al. Health monitoring of civil infrastructures using wireless sensor networks. In *IPSN*, Apr. 2007.

[8] K. Klues et al. Integrating concurrency control and energy management in device drivers. In *SOSP*, 2007.

[9] N. G. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, July 1993.

[10] G. Mainland, G. Morrisett, and M. Welsh. Flask: Staged functional programming for sensor networks. In *Proc. 13th ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP)*, 2008.

[11] T. Murata. Petri nets: Properties, analysis and applications. *Proc. of the IEEE*, 77(4), 1989.

[12] Y. Nir-Buchbinder, R. Tzoref, and S. Ur. Deadlocks: from exhibiting to healing. In *Wkshp on Runtime Verification*, Mar. 2008.

[13] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1), 1987.

[14] N. Ramanathan et al. Sympathy for the sensor network debugger. In *SenSys*, 2005.

[15] R. Sengupta. Diagnosis and communication in distributed systems. In *Wkshp on Discrete Event Systems*, Aug. 1998.

[16] H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7), Sept. 2005.

[17] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *OSDI*, Dec. 2008.

[18] Y. Wang, T. Kelly, M. Kudlur, S. Mahlke, and S. Lafortune. The application of supervisory control to deadlock avoidance in concurrent software. In *Wkshp on Discrete Event Systems*, May 2008.

[19] Y. Wang, T. Kelly, and S. Lafortune. Discrete control for safe execution of IT automation workflows. In *EuroSys*, 2007.

[20] Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, and S. Mahlke. The theory of deadlock avoidance via discrete control. In *POPL*, Jan. 2009.