Report on the

Third Workshop on Hot Topics in Software Upgrades (HotSWUp'11)

http://www.hotswup.org/2011/

Christopher M. Hayden
Dept. of Computer Science
University of Maryland
College Park, MD
hayden@cs.umd.edu

Iulian Neamtiu
Dept. of Computer Science and Engineering
University of California, Riverside
Riverside, CA
neamtiu@cs.ucr.edu

ABSTRACT

The Third Workshop on Hot Topics in Software Upgrades (HotSWUp'11) was held on April 16, 2011 in Hannover, Germany. The workshop was co-located with ICDE 2011. The goal of HotSWUp is to identify, through interdisciplinary collaboration, cutting-edge research ideas for implementing software upgrades.

The workshop combined presentations of peer-reviewed research papers with a keynote speech on the practical issues related to performing large-scale upgrades. The audience included researchers and practitioners from academia, industry, and government. In addition to the technical presentations, the program allowed ample time for discussions, which were driven by debate questions provided in advance by the presenters.

HotSWUp provides a premier forum for discussing problems that are often considered niche topics in the established research communities. For example, the technical discussions at HotSWUp'11 covered dynamic software updates, package management tools, database schema upgrades, upgrades of systems with real-time constraints, upgrading satellite software, and highlighted many synergies among these and other topics.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.3.3 [**Programming Languages**]: Language Constructs and Features; D.4.7 [**Operating Systems**]: Organization and Design; H.2.1 [**Database Management**]: Logical Design; K.6.3 [**Management of Computing and Information Systems**]: Software Management

General Terms

Management, Experimentation, Human Factors, Performance, Reliability

Keywords

Software upgrades, dynamic software update, package management, database schema evolution, real-time upgrades

HotSWUp'11 Overview

The HotSWUp'11 program featured three research sessions and a discussion period for each. Prior to the sessions, Philip Bernstein gave a keynote describing an approach to automating schema and object-relational-mapping evolution. The first session included work related to software update correctness; the second covered topics related to database evolution; and the final session covered systems for runtime updating. This report summarizes the presentations and discussion that took place during each session.

Keynote Address

Schema and Mapping Evolution in an Object-Relational Mapper by *Philip Bernstein (Microsoft Research)*

In the keynote address, Philip Bernstein discussed the problem of modifying a database's schema and object-relational mapping (ORM) to reflect changes to a program's entity types (i.e., the classes representing persistent data). The keynote talk began with an overview of the variety of ORM strategies that are used by real-world programs, which motivated the need for tools to facilitate evolution and also presented challenges to automating evolution. Next, Bernstein discussed two related research problems that he and his collaborators have pursued: (1) an approach to inferring the mapping patterns in use to allow the ORM and schema to be modified automatically, and (2) avoiding full recompilation of an ORM mapping that has evolved.

Updating the schema and ORM in response to entity type changes is challenging because there are a variety of ORM patterns that can be used and the patterns for a particular program may be unknown or even heterogeneous. Bernstein described three common mapping patterns:

- Table-per-type - where a separate relational table is created for each abstract and concrete persistent object class in the program,
- Table-per-concrete-type - where all of the data for a particular concrete type is stored in a single table, and
- Table-per-hierarchy - where many related concrete types are all stored within a single table.

Further, there are many ways to allocate/reuse columns for each entity type in the table per hierarchy approach. To update the schema and ORM when an entity type changes, it is necessary to determine the mapping pattern that applies to the types/attributes that were affected.

Bernstein described an approach they developed to automate this process, wherein each mapping from entity/attribute to table/column is treated as data and mined to determine the mapping patterns in use [24, 25]. The mapping patterns that affect a particular entity type are inferred based on the notion of "local scope" which is computed by applying a cutoff to distance metrics (e.g., sibling classes are closer than ancestors). Bernstein then showed that, given an inferred mapping pattern, they can support entity addition, removal, modification, and refactoring (i.e., changes to the class hierarchy). This approach was implemented as an extension to Visual Studio. This tool updates the mapping and store model automatically and supports directly updating the SQL database or generating a script to apply the update at deployment.

Finally, Bernstein described a technique to incrementally compile the mapping specification into executable views, thereby avoiding the expense of a recompiling the full mapping specification. Full recompilation is prohibitive for large models because it involves an expensive test to ensure that the ORM correctly round-trips.

Together, these two techniques allow developers to modify entity types and have the system generate appropriate modifications to the schema and ORM.

Discussion

Following the talk, an audience member asked whether these problems could be handled more easily if the mapping patterns were manually specified, rather than inferred. Bernstein responded affirmatively, and noted that this has been done somewhat in the past, for example with annotations on the entities. However the keynote's research targets real-world systems where these mappings are not specified.

The audience asked whether schema changes applied using this system to a "hacky" entity-database mapping might make the mapping worse. Bernstein acknowledged this concern, but noted that performance often motivates inelegant ORM choices, so "hacky" mappings may be unavoidable.

The audience asked whether problems associated with evolving mappings from objects to relational tables could be improved by just using a database with object-oriented features. The speaker pointed out that many DBMSs do not have OO features.

The audience asked whether it was also important to consider the speed of performing the migration rather than just the speed of computing/updating the mapping, in order to support on-line migration. The speaker noted that on-line database migration is a hard problem because temporary tables are required for many types of updates.

The audience asked whether the information gleaned from inferring the entity-db relationships could be used to enable advanced, vendor-specific features, like sparse columns. Bernstein noted that they had not considered this, but that it might be useful.

Session 1: Update Semantics and Analysis

Formal Reasoning about Runtime Code Update by *Nathaniel Charlton, Ben Horsfall, Bernhard Reus (University of Sussex)* [4]

Nathaniel gave the talk. He presented an approach to verifying the correctness of dynamic software updates. At a high level, the approach aims to construct proofs about the safety of runtime updates using Hoare logic. The problem with using standard Hoare logic for correctness is that pre- and post- conditions only reason about data, not code. To support reasoning about code changes, the authors use *nested Hoare triples*—assertions that characterize the behavior of a procedure w.r.t. how it manipulates the heap.

The authors propose a strategy that requires developers to write their program such that it contains a transition function that installs new code and data. For a program written in this way, Nathaniel described how to verify updates using a tool called Crowfoot that performs verification based on nested Hoare triples. This technique was used to verify updates to a web-server example given in the paper Formalizing Dynamic Software Updating (Bierman, Hicks, Sewell, Stoyle)[2]. In the presented work, the authors studied the safety of adding logging to a web server.

An audience member asked whether this technique would allow checking higher-level properties referring to state that persists across the update. Nathaniel responded that such properties are hard to prove using (nested) Hoare triples. Michael Hicks noted that implementing program updates as a program+transition function mirrors the compilation strategy used by many DSU compilers and asked whether that particular approach was selected for compatibility with nested Hoare triples. Nathaniel responded that he had evidence that these proofs were possible with other verification strategies.

An audience member remarked that users of Crowfoot manually do the job of an update compiler (e.g., indirection, function pointers) and asked whether this was the intent of the design. Nathaniel answered that what is essential is modeling and verifying the effects of updating on the heap.

An audience member asked whether the authors could prove whether a certain behavior that was present in the old version is preserved in new version. Nathaniel noted that their system can not prove program equivalence.

Towards a Categorical Framework to Ensure Correct Software Evolutions by *Sylvain Bouveret, Julien Brunel, David Chemouil, and Fabian Dagnat* [3]

David gave the talk. He began the talk by presenting the motivation behind their work: verification of satellite control software. For example, one satellite software requirement might be flying in formation, hence when patches are applied to individual satellites, what is the impact of the upgrade on the whole formation? Their approach is to find out the proof obligations for requirements using a categorical framework (correctness by construction, not a posteriori). Category theory is well-suited for representing abstract architectures (since, in category theory, objects are black box elements). The application architecture is first mapped to an *abstract category* (in-out relationship between objects) and then to a *concrete category* (contracts in LTL [22] formulas). Patches can contain several elements, e.g., renaming a proposition, removal/additions of axioms, or combining elements of similar nature into one. They leave several items to future work: declarative language for patches, assess generation of proof obligations in practice.

One audience member asked that, since elements have no state, isn't the approach actually addressing software evolution, rather than dynamic updates. David answered that indeed state is not taken into account.

Predicting Upgrade Failures Using Dependency Analysis by *Pietro Abate and Roberto Di Cosmo* [5]

Pietro gave the talk. Their work seeks to determine the critical software packages in distributions of programs and libraries for Linux-based operating systems. Their objective is to identify "important" packages based on metadata (e.g., by looking at conflicts and dependencies), categorize packages based on their importance, and also predict the packages for which upgrades are most risky. Their approach is to first compute the *strong impact set* for each component p, which consists of the packages that must be installed for p to be installed. The next step is to compute a *prediction map* for p—packages that will surely be broken when p is upgraded. They have implemented a prediction model to determine what future updates to packages would break the most other packages and have applied it to Debian. The results show that GCC, libc, and Perl are among the packages with large impact sets and prediction maps. Next, they cluster packages that should be upgraded together, e.g., GCC and libgcc and found that when all the components in a cluster are updated together, no other packages are broken. In the future they plan to formalize this approach, and apply to other package managers (rpm worlds, Eclipse).

Discussion

Iulian Neamtiu noted that he sees a dichotomy in the approaches taken: the first approach is operational and the second is behavioral. Michael Hicks made another high-level observation that we care about both individual software correctness and architecture; also, he asked whether we should consider behavior that is removed as a result of the update.

An audience member noted that CROWFOOT is not specifically written to work with dynamic updates and wondered whether other verification techniques might be similarly useful.

Rida Bazzi asked what kind of semantic changes should be allowed, e.g., if the update changes a "bad behavior" to a fail-stop behavior, should it be allowed? Michael Hicks thinks this should probably be supported. Rida added that sometimes the application vendor wants to make sure a certain behavior is gone, e.g., a vulnerability is eliminated, whereas sometimes it is desirable to crash (e.g., crash when a vulnerability is triggered, rather than continue).

Carlo Zaniolo asked whether employing testing techniques might address some of the weaknesses of DSU verification techniques and wondered whether the two could be combined. Michael Hicks pointed out that Hayden et al.'s paper at HotSWUp'09 [13] provides an approach for testing dynamic updates. Michael Wahler (ABB Research) mentioned that an ongoing EU project, PINCETTE [21], is integrating static and dynamic analysis techniques to identify the impact of intra-component changes and component replacement. Rida mentioned that their work at Arizona State University is also investigating this problem. Carlo Zaniolo asked why is there no work in testing application updates, e.g., using simulation to assess the impact of an update to a cloud application.

The discussion then moved to the subject of online upgrades of databases. An audience member asked whether some of this correctness work is applicable to database changes. Philip Bernstein noted that not many people are performing on-line migration since it is perceived as too dangerous, but correctness of rolling updates may be an interesting problem to address formally.

One audience member commented that supporting general online database upgrades is "scary" for practitioners, but certain schema changes can safely be performed online. For example, adding a column is feasible, but splitting a table may be more dangerous, or slow. Philip Bernstein remarked that it would be useful to have some automated tools to determine the dependencies within a database, as dependencies would help indicate the order in which tables should be upgraded; he also pointed out that in-place migration is not widely used as it is considered too dangerous.

Michael Hicks remarked that the dependency analysis from Abate and di Cosmo's work would be useful in finding out which packages can be safely upgraded depending on who is logged on and what programs they are running. Pietro Abate pointed out that, right now, sysadmins never apply updates with users logged on, so, their system would indeed be helpful in allowing updates to be applied without kicking users out first. Another audience member remarked that the dependency work can also be applied to impact of schema evolution, by performing a translation from DB/schema evolution terminology into the dependency framework.

Session 2: Database Upgrades

Schema Evolution Analysis for Embedded Databases by *Shengfeng Wu and Iulian Neamtiu* [27]

Iulian Neamtiu gave the talk. The motivation for their work was the ubiquity of embedded databases (e.g., SQLite), and the observation that, to support dynamic updates to applications, online schema upgrades must be supported as well. To understand how embedded database' schemas evolve, he described an empirical study of the evolution of embedded databases for programs that persist data using SQLite. This work compared empirical results against prior studies that looked at evolution of database schemas for server programs [6, 16]. The presenter provided two observations: database schemas change mostly in the beginning of an application's evolution; and changes are usually simple, with column additions, deletions, and type changes being most frequent.

An audience member noted that the study compared applications that were quite different and suggested that a different selection of subject applications might have produced different results. Iulian Neamtiu pointed out that because the study looked at a wide range of programs— both desktop and server applications—the results should generalize. Another audience member observed that the study compared changes to databases with changes to program code and suggested that looking at data structure changes in the program might have produced a stronger correlation; the presenter agreed.

Causes for Dynamic Inconsistency-tolerant Schema Update Management by *Hendrik Decker* [11]

This talk discussed the observation that databases are often able to support querying, updating, and other reasoning even when the facts stored in the database become inconsistent. The work presented definitions for integrity constraints and database updates using Datalog, as well as examples of various database updates that violate or preserve integrity constraints. Prior work by the author [9, 8] showed that inconsistency-tolerant integrity-checking methods preserve *cases*, i.e., instances of integrity constraints satisfied prior to the update. In later work [7], they considered preventing an increase in *causes* of inconsistency (i.e., database facts responsible for integrity violations). Here, Hendrik described how cause-based inconsistency-tolerance can also be ensured for updates that change integrity constraints. He also showed that each cause-based method is also case-based and shares the advantages of the case-based approaches.

An audience member asked whether this approach supported negation. The presenter responded affirmatively and noted that negations are needed to produce inconsistencies.

Propagating Evolution Events in Data-Centric Software Artifacts by *George Papastefanatos, Panos Vassiliadis, and Alkis Simitsis* [20]

George gave the talk. This work seeks to provide an approach for determining which parts of a database are affected by a schema change. To do this, the authors present a view of a data management system as an *Architecture Graph* that connects database tables, schemas, constraints, and queries. The system determines how a database change will affect the nodes of the graph by propagating notification of the change as messages between nodes in the graph. The "status" of each node is determined based on the messages received by that node and its policies. A node's status indicates whether a change affects that node and nodes may reject certain changes that are incompatible with installed policies. The authors prove termination and correctness of the propagation process in their paper.

An audience member asked why it was useful to propagate the effects of schema changes throughout a data model when there will be other parts of the system (e.g., the program code) that depend on the particular views and queries that were previously in place. The presenter responded that this approach allows the developer to understand and automate the adjustment of the affected views.

Another audience member asked whether this approach will produce node statuses deterministically. The presenter noted that their previous work did not guarantee determinism, but this work does since a node's status is only computed after all of its inputs have been computed.

Discussion

An audience member asked Iulian Neamtiu how often he has observed "regression" in database schemas where changes that have been added are subsequently undone. Neamtiu indicated that he observed this only a few times out of hundred of commits they have investigated.

An audience member noted that facilitating schema evolution was a popular research topic, and asked why there has been little take-up of these techniques by industry. This elicited a good discussion from both the speakers and the audience about this question, as described next.

One audience member noted that the theory presented in the research world was quite clean, but databases in the real world are much more messy, giving the example of consistency checking as a tool that sometimes fails to work with real-world systems. Another audience member agreed, observing that users disable consistency checking because it is too expensive. Another audience member suggested that the lack of interest might result from users caring more about features they can use right away than tools that will help them maintain their software in the future. Carlo Zaniolo noted that schema evolution techniques had not begun to come into their own until recently, suggesting that the techniques might be approaching sufficient maturity for adoption.

Philip Bernstein suggested that the rapid pace of DB runtime innovation makes it difficult for tools for schema evolution to keep pace. Further, he noted ISVs have a tough time keeping up with new features and may not have resources to expend on schema evolution tools. However, he noted that, if one vendor "gets the ball rolling," this might lead to other vendors following; he also noted that universities could be in a good position to build an open source migration solution—and suggested that it might turn out to be hard problem!

Another audience member pointed out that Oracle Flashback allows people to go to previous schema versions, suggesting there is hope that such tools will be adopted. Once

companies realize there is a need for automatic support for managing the schema history, then there will be more of a market for these tools.

Rida Bazzi noted that Facebook has published some basic software for supporting online evolution of schemas. He noted that this is an important problem since some critical applications (like hospital software systems) might only update once a year and cannot even tolerate two hours of downtime, due to policy or regulation.

An audience member wondered whether cloud computing would be a driver for schema evolution uptake—for example, Facebook implements their schema updates in application code [10]. Philip Bernstein suggested that the cloud requires gradual roll-out of data changes over a large number of machines and distributed processes; the main challenges in this context are with the program code and not the schema itself.

Another audience member noted that cloud computing changes our assumptions significantly due to its distributed nature and that might provide new ways of thinking about migration since many traditional database consistency guarantees need to be relaxed. Iulian Neamtiu agreed, noting that many large scale systems, like YouTube, could not operate if strict ACID guarantees were required.

Session 3: Approaches and Systems

Hot-updates for Java-based Smart Cards by *Agnes Cristele Noubissi, Julien Iguchi-Cartigny and Jean-Louis Lanet* [19] (invited paper)

Agnes gave the talk. She started by observing that smart cards are widely used for SIM cards, bank cards, and electronic passports, and periodically, the software on the smart card needs to be updated (e.g., due to new banking regulations or travel security requirements). Their solution to smartcard updates is called EmbedDSU: an embedded JVM with DSU support. Their system consists of a *DIFF Generator* that is run prior to deployment to the smart card, and a custom JVM running on the card to apply the update.

The first step of this approach is the execution of the *DIFF Generator*, which calculates the differences between the old and new code, and produces output in a DSL called DIFF. Once the DIFF patch has been created, it can be uploaded to a running smart card using a card reader. When an update request occurs, the card stores the DIFF in persistent memory and waits until all active stack frames corresponding to modified methods have returned. When a safe point is reached, EmbedDSU updates class metadata including the constant pool, field table, and method table for each modified class. To update the instances of changed classes to their new representation, a modified garbage collector traverses the heap performing transformation.

An audience member asked what motivated this work. Agnes answered that a certain telecom company wants to support updating in their SIM cards. She also noted that one constraint that differentiated this work from other DSU systems was the need to take care to minimize energy consumption. Another audience member asked what steps are taken to guarantee that updates to passports or other critical smart cards are secure. Agnes answered that EmbedDSU requires patches to be signed to guarantee integrity.

Non-disruptive Large-scale Component Updates for Real-Time Controllers by *Michael Wahler, Stefan Richter, Sumit Kumar, Manuel Oriol* [26]

Michael gave the talk. The primary motivation for this work is that many mission-critical systems that need to be continuously available have real-time constraints. Therefore, they must support updating real-time system components while preserving state and respecting time bounds. Michael described a typical real-time application with a 5 ms cycle. To be on the safe side, loading new components must complete in about 2/3 of a cycle time; the rest is "slack time." The challenge is transferring state to the new components: in particular, it may not be possible to complete the transfer within a single cycle.

The authors' solution is to break down the transfer so that it can be performed over multiple cycles. To support this, they use a dirty bit vector to store the state transfer status for each component; the transfer is known to be completed when all bits are marked as "clean."

An audience member asked whether there is an implicit assumption that the updating process converges. Michael agreed that they make this assumption, and it seems that in practice their scheme converges; however, there is no explicit convergence guarantee.

Another participant asked why there are no global variables in this approach. Michael answered that, since each component is in a different address space (for dependability reasons), there are no global variables.

Another participant asked whether the slack time percentages should be even greater, e.g., to allow for failures. Michael answered that, even though in their experience 1/3 slack time was sufficient, he could foresee cases where more than a 1/3 would be needed.

State Transfer for Clear and Efficient Runtime Upgrades by *Christopher Hayden, Edward Smith, Michael Hicks, and Jeffrey Foster (University of Maryland, College Park, USA)* [12]

Christopher gave the talk. He began by pointing out that many DSU tools have been proposed: Ginseng [18, 17], POLUS [23], UpStare [15], JVolve [23], but they suffer from several limitations: reasoning about post-update behavior (e.g., ensuring that the update is safe or correct), steady-state performance overhead (e.g., tools such as Ginseng use a special compilation scheme that leads to steady-state performance overhead even in the absence of updates); and use of nonstandard tools (e.g., special compilation).

The authors' solution is Ekiden, a DSU system for C programs that updates a program by starting the new version from scratch and transferring the state from the running version. With Ekiden, the programmer prepares programs

for updating by "tagging" state that needs to be transferred, adding explicit update points to long-running loops, adjusting program flow to return to the correct loop following an update, and writing transformation functions to convert old-version state to work with the new version. The Ekiden library provides a set of tools for writing C programs that support state-transfer DSU.

Ekiden addresses many of the limitations of existing DSU systems: updated program behavior is explicit in the program code and therefore easier for the developer to reason about, updating incurs no steady-state overhead because no costly indirection must be compiled into the program, and updating can be implemented as library rather than requiring special compilation.

One participant asked which platforms Ekiden supports. Christopher answered that they had currently tested the implementation on GNU/Linux and Mac OS X. Iulian Neamtiu asked whether there might be program state that cannot be transferred using Ekiden, such as the process PID, parent-child relationships among processes or a socket's state. The speaker answered that those types of state can be transferred. For example, file descriptors and sockets are preserved because the new version is started via fork() + exec(); however, transferring internal library state may require extra effort.

Discussion

One participant asked the session panelists to name some reasons why customers are not eager to embrace DSU. Michael Wahler answered that, in many cases, customers want software safety certification (e.g., for nuclear power plants); in his view, software certification for DSU is a long way off. Another suggested explanation was that programmers may worry more about single version features than supporting smooth evolution in the future. Christopher Hayden observed that DSU is a feature targeted to users/administrators, who may begin to demand it as more programs begin to provide support. He noted that DSU is different from a low-level runtime feature like garbage collection that is targeted at programmers.

Michael Wahler was asked whether formalization techniques presented in the earlier sessions would be useful for real-time systems. His answer was that many real-time software developers are electrical engineers, not computer scientists, and may lack a formal methods background. In addition, he noted that extensive testing is the current standard and that it may be impractical to require a full formal specification of the system.

One participant asked about the current certification process for space industry projects. The answer was that these projects use a single process (e.g., model-driven engineering that integrates model design with model verification—source code is automatically generated from the model).

One participant asked how to guarantee update safety for nuclear power plants. Michael Wahler suggested the use of redundant systems as a solution. Iulian Neamtiu pointed out that redundancy is not always the answer, and may introduce new problems, as in the 2008 nuclear power plant shutdown incident where an update installed on a single computer in the monitoring system led the monitor to disagree with the control system causing the plant to shut down [14].

An audience member asked whether certifying DSU patches would mean showing that dynamic and offline upgrades achieve the same result. The answer was no, because definitions of DSU correctness must specify correct handling of in-memory state, which is not maintained in offline upgrades.

One participant wondered whether DSU could be popularized by making it available on Linux. Another participant noted that Ksplice currently supports dynamic updates to the Linux kernel [1].

Acknowledgments

We thank the program chairs: Michael Hicks, Rida A. Bazzi, Carlo Zaniolo. We also thank the program committee: Carlo Aldo Curino, Fabien Dagnat, Johann Eder, Manuel Oriol, George Papastefanatos, Paolo Papotti, Jason Nieh, Mark Segal, Liuba Shrira, Xin Qi. Finally, we would like to thank the authors of submitted and invited papers for providing the excellent content of the program and for their enthusiastic participation in the workshop.

1. REFERENCES

[1] J. Arnold and M. F. Kaashoek. Ksplice: automatic rebootless kernel updates. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 187–198, New York, NY, USA, 2009. ACM.

[2] G. Bierman, M. Hicks, P. Sewell, and G. Stoyle. Formalizing dynamic software updating. In *On-line Proceedings of the Second International Workshop on Unanticipated Software Evolution (USE)*, pages 13–23, 2003.

[3] S. Bouveret, J. Brunel, D. Chemouil, and F. Dagnat. Towards a categorical framework to ensure correct software evolution. In *HotSWUp '11: Proceedings of the Third Workshop on Hot Topics in Software Upgrades*, Hannover, Germany, 2011.

[4] N. Charlton, B. Horsfall, and B. Reus. Formal reasoning about runtime code update. In *HotSWUp '11: Proceedings of the Third Workshop on Hot Topics in Software Upgrades*, Hannover, Germany, 2011.

[5] R. D. Cosmo and P. Abate. Predicting upgrade failures using dependency analysis. In *HotSWUp '11: Proceedings of the Third Workshop on Hot Topics in Software Upgrades*, Hannover, Germany, 2011.

[6] C. Curino, H. J. Moon, L. Tanca, and C. Zaniolo. Schema evolution in wikipedia - toward a web information system benchmark. In *ICEIS (1)*, 2008.

[7] H. Decker. Toward a uniform cause-based approach to inconsistency-tolerant database semantics. In *Proceedings of the 2010 international conference on On the move to meaningful internet systems: Part II*, OTM'10, pages 983–998, Berlin, Heidelberg, 2010. Springer-Verlag.

[8] H. Decker and D. Martinenghi. Classifying integrity checking methods with regard to inconsistency

tolerance. In *Proceedings of the 10th international ACM SIGPLAN conference on Principles and practice of declarative programming*, PPDP '08, pages 195–204, New York, NY, USA, 2008. ACM.

[9] H. Decker and D. Martinenghi. Inconsistency-tolerant integrity checking. *IEEE Transactions on Knowledge and Data Engineering*, 23:218–234, 2011.

[10] T. Dumitraş, I. Neamtiu, and E. Tilevich. Second acm workshop on hot topics in software upgrades (hotswup 2009). In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 705–706, New York, NY, USA, 2009. ACM.

[11] C. for Dynamic Inconsistency-tolerant Schema Update Management. Hendrik decker. In *HotSWUp '11: Proceedings of the Third Workshop on Hot Topics in Software Upgrades*, Hannover, Germany, 2011.

[12] C. Hayden, E. Smith, M. Hicks, and J. Foster. State transfer for clear and efficient runtime upgrades. In *HotSWUp '11: Proceedings of the Third Workshop on Hot Topics in Software Upgrades*, Hannover, Germany, 2011.

[13] C. M. Hayden, E. A. Hardisty, M. Hicks, and J. S. Foster. Efficient systematic testing for dynamically updatable software. In *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades*, HotSWUp '09, pages 9:1–9:5, New York, NY, USA, 2009. ACM.

[14] B. Krebs. Cyber Incident Blamed for Nuclear Power Plant Shutdown. *Washington Post*, June 5, 2008. `http://www.washingtonpost.com/wp-dyn/content/article/2008/06/05/AR2008060501958.html`.

[15] K. Makris and R. Bazzi. Multi-threaded dynamic software updates using stack reconstruction. In *USENIX ATC*, 2009.

[16] V. Mandalapa. A framework for understanding schema evolution in web information systems. Master's thesis, Arizona State University, 2009.

[17] I. Neamtiu and M. Hicks. Safe and timely updates to multi-threaded programs. *SIGPLAN Not.*, 44:13–24, June 2009.

[18] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for c. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 72–83, New York, NY, USA, 2006. ACM.

[19] A. C. Noubissi, J. Iguchi-Cartigny, and J.-L. Lanet. Hot updates for java-based smart cards. In *HotSWUp '11: Proceedings of the Third Workshop on Hot Topics in Software Upgrades*, Hannover, Germany, 2011.

[20] G. Papastefanatos, P. Vassiliadis, and A. Simitsis. Propagating evolution events in data-centric software artifacts. In *HotSWUp '11: Proceedings of the Third Workshop on Hot Topics in Software Upgrades*, Hannover, Germany, 2011.

[21] PINCETTE Project. `http://pincette-project.eu/`.

[22] A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46 –57, 31 1977-nov. 2 1977.

[23] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic software updates: a vm-centric approach. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 1–12, New York, NY, USA, 2009. ACM.

[24] J. F. Terwilliger, P. A. Bernstein, and A. Unnithan. Automated co-evolution of conceptual models, physical databases, and mappings. In *Proceedings of the 29th international conference on Conceptual modeling*, ER'10, pages 146–159, Berlin, Heidelberg, 2010. Springer-Verlag.

[25] J. F. Terwilliger, P. A. Bernstein, and A. Unnithan. Worry-free database upgrades: automated model-driven evolution of schemas and complex mappings. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 1191–1194, New York, NY, USA, 2010. ACM.

[26] M. Wahler, S. Richter, S. Kumar, and M. Oriol. Non-disruptive large-scale component updates for real-time controllers upgrade. In *HotSWUp '11: Proceedings of the Third Workshop on Hot Topics in Software Upgrades*, Hannover, Germany, 2011.

[27] S. Wu and I. Neamtiu. Schema evolution analysis for embedded databases. In *HotSWUp '11: Proceedings of the Third Workshop on Hot Topics in Software Upgrades*, Hannover, Germany, 2011.