Dependability Gauges for Dynamic Systems

R. A. Riemenschneider and Joshua Levy
System Design Laboratory
SRI International
333 Ravenswood Av
Menlo Park, CA 94025
Telephone: 650-859-{2507,2868}
Fax: 650-859-2844
{rar,levy}@sdl.sri.com

***Abstract*— This paper introduces *dependability gauges*, which measure dependability properties of dynamically evolving systems. The general concept is illustrated by explaining the details of a fault tolerance gauge that we have implemented.**

I. Introduction

It will soon be the case that most systems will be constructed, at least in part, from pre-existing components. The infrastructure needed to support a component-based lifecycle is currently emerging: intercomponent communication mechanisms (CORBA, DCOM) and data interchange formats (XML, DOM), service discovery mechanisms (Jini, e-Speak), and even higher-level collaboration and delegation mechanisms (SRI's Open Agent Architecture).

But a component-based lifecycle also poses new software engineering challenges. Most components developed for the commercial market will not be developed with the high dependability requirements of, e.g., DoD mission-critical applications in mind. So, if developers of highly dependable systems are to take maximal advantage of the availability of components, one question that must be answered is: *How can a highly dependable system be built from components that may not be dependable?*

Basing systems on components will also increase the pace of system evolution. Components will quickly be declared obsolete, and replaced by new versions. As new versions of components, offering new capabilities, become available, users will naturally want to exploit those capabilities. Other pressures driving evolution — for example, the need to respond to changes in missions — can only intensify as well. Thus, another question that must be answered is: *How can dependability be maintained when a system is constantly evolving?*

Previous SRI research on the design and construction of architectures for secure distributed transaction processing has shown how it is possible to build a secure system from not-necessarily-secure components. The primary innovation in our approach is to link an abstract architectural model that is proven secure to the implemented system architecture by a series of transformations that demonstrably preserve security. This link allows us to conclude that results obtained from our security analysis of the abstract model are applicable to the implementation as well. The same technique can be used to establish other system dependability properties.

The research described in this paper was supported by the Defence Advanced Research Projects Agency of the U.S. Government under contract F30602-00-C-0199 with the Air Force Research Laboratory.

We are building upon this previous research, first, by generalizing our approach to dependability properties other than security and, second, by using the transformation chains that link the abstract, analyzable architectural models — one per dependability property — to the concrete system architecture to dynamically update the abstract models as the running system evolves, making it possible to build *dependability gauges* that continuously measure dependability properties of an evolving system.

This paper will first describe in more detail what dependability gauges are and how they work, and then describe the first dependability gauge we have implemented.

II. Designing for Dependability

There are two approaches to guaranteeing that a system satisfies a dependability constraint. The difference between these approaches can be illustrated by a simple example. Consider a system composed from components with varying security levels and clearances. Security policies for such multilevel systems include constraints on communication. For example, a typical constraint is that "read-up" is not allowed. That is, a component must not read data whose classification is greater than the component's clearance. How can we guarantee that such constraints are satisfied if the system contains closed-source components whose security has not been verified? One approach is to monitor all communication among components at runtime, and check whether the security policy is satisfied in each case. In cases where the communication would violate the security policy, the communication is blocked. An alternative approach is to design a system architecture that restricts communication among components so as to reduce the need for runtime constraint checking. If the architecture is designed so that communication channels between components exist only when communication between those components is consistent with the system's security policy, then no runtime checking of the security constraints is needed.

Our research has explored the latter approach. One product of that research is SDTP [Moriconi, *et al.* 97; Gilham, Riemenschneider & Stavridou 99], a dynamic architecture for secure distributed transaction processing. SDTP was designed by writing a simple, abstract description of the architecture, showing that the description guarantees the desired security properties, successively refining the abstract description until a directly implementable concrete description results, and showing that each refinement step preserves satisfaction of the security policy. Refinement steps were the result of applying reusable refinement transformations that codify implementation techniques. Thus, SDTP is an example of how a dependable (in this case, secure) system can be constructed from not-necessarily-dependable components, without the overhead of runtime constraint checking.

III. Generating Dependability Gauges by Abstraction

Formal methods, such as model checking and theorem proving, can be effective for determining whether systems have desired dependability properties. Less formal methods, such as simulation, also provide useful information about dependability, even when incapable of providing definitive answers. But, for complex systems, these methods cannot be applied directly. An abstract system model, designed specifically for the purposes of the particular dependability analysis, must be created. As much system detail as possible is abstracted away, in order to make the analysis more tractable.

Model checking fault tolerance of a system provides a good illustration of the necessity of abstraction. Fault tolerance means that no combination of system state transitions and failure transitions can lead to a state where a system cannot supply essential services. Naïve models of complex systems will have too many states — often, infinitely many — for exhaustive state exploration to be feasible. This problem is particularly acute when closed-source components are involved, since such components' states potentially depend upon the entire history of their interactions with the system. Of course, for a fault tolerance analysis, a very simple model of such components is sufficient. Only the component's external interface protocol and whether it is faulty or nonfaulty is relevant. But, when a system is constructed from a large number of components with complex interface protocols in accordance with a complex architectural description, the number of states may still be too large for model checking. Boolean abstraction of protocols proves very useful in such cases, and SRI has developed techniques for automatically determining relevant predicates for use in the abstraction [Graf & Saïdi 97; Saïdi & Shankar 99].

In prior work, our application of abstraction techniques has been at design time. In the component-based lifecycle, where

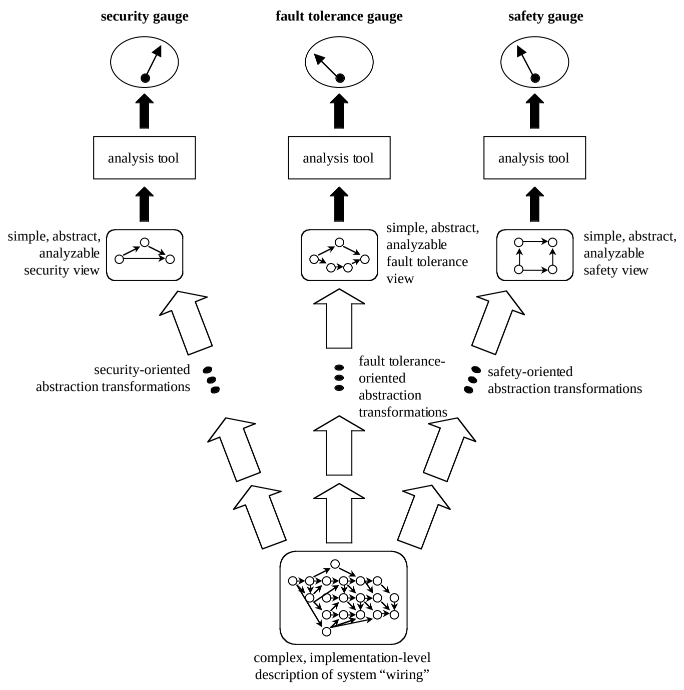

security gauge
fault tolerance gauge
safety gauge
analysis tool
analysis tool
analysis tool
simple, abstract, analyzable security view
simple, abstract, analyzable fault tolerance view
simple, abstract, analyzable safety view
security-oriented abstraction transformations
fault tolerance-oriented abstraction transformations
safety-oriented abstraction transformations
complex, implementation-level description of system "wiring"

Fig. 1. Dependability Gauges

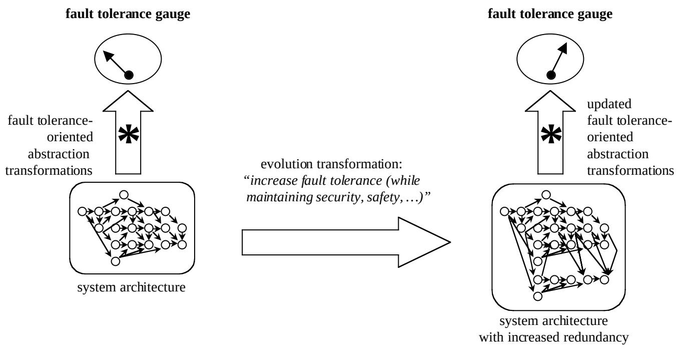
the system is constantly evolving, there are obvious advantages to applying them at runtime as well. For each dependability property of interest, an abstract system model that can be analyzed to determine whether the system has the property will be generated. The abstraction transformation steps will demonstrably co-preserve dependability. In other words, if the generated abstract model is dependable, then the more concrete input to the abstraction step must be dependable as well. As the system evolves, the abstract models will be updated by "replaying" the derivations that generated them. Thus, the dependability of the system can be dynamically reassessed whenever there are relevant changes in the environment or within the system itself. The results of analysis of the abstractions are displayed as dependability gauge readings, as shown in Figure 1.

IV. Dependable Evolution through Perpetual Design

Much as transformations can be used at design time to create a baseline architecture that has desired dependability properties, transformations can be used at runtime to ensure continued dependability as requirements, the system, and its environment evolve. The basic idea is that changes in dependability gauge readings can trigger application of dependability-enhancing, functionality-preserving architecture transformations.

Any of the following may trigger re-architecting.

- *A change in required dependability level.* The system administrator may decide that the system should be made more (or less) secure, more (or less) fault tolerant, more (or less) safe, etc. Architectural transformations expected



fault tolerance gauge
fault tolerance-oriented abstraction transformations
system architecture
evolution transformation:
"increase fault tolerance (while maintaining security, safety, ...)"
fault tolerance gauge
updated fault tolerance-oriented abstraction transformations
system architecture with increased redundancy

Fig. 2. Re-Architecting to Improve Fault Tolerance

to have the appropriate effect will be applied, and the results monitored using the appropriate dependability gauge, as shown in Figure 2. Other dependability gauges will be monitored as well, to ensure that other dependability constraints remain satisfied.

- *A change in desired functionality.* The system user may request that the system provide some additional service, requiring the integration of additional components or the replacement of components with other components offering additional functionality. If a new component is added to the system, or one component is replaced by another that proves less dependable, then the system architecture may have to be made more dependable to compensate. An example is provided by our SDTP work, where we showed that a very simple architecture can be used to securely integrate single-level databases that are all at the same level, but substantial architectural complexity must be introduced when a database at a different level is integrated into the system.
- *A change in component availability.* Changes in component availability can also necessitate addition and replacement of system components, and hence changes in system architecture.
- *A change in the system's behavior or performance.* Monitoring of dependability gauges may reveal dependability requirements that were being satisfied but are no longer being satisfied. Similarly, monitoring of performance gauges may reveal that system performance has declined to an unsatisfactory level. Architectural transformations can be applied to address such problems.

V. Dependability Gauges for Fault Tolerance

The first dependability gauge suite we have implemented measures a simple fault tolerance property. We suppose that a system's requirements identify some of the functions performed by the system as critical, and mandates that, for each critical function, some minimum number of failures of components that contribute to that function must be tolerated without loss of functionality. Figure 4 contains a archiectural description of such a system, in the Acme architectural description

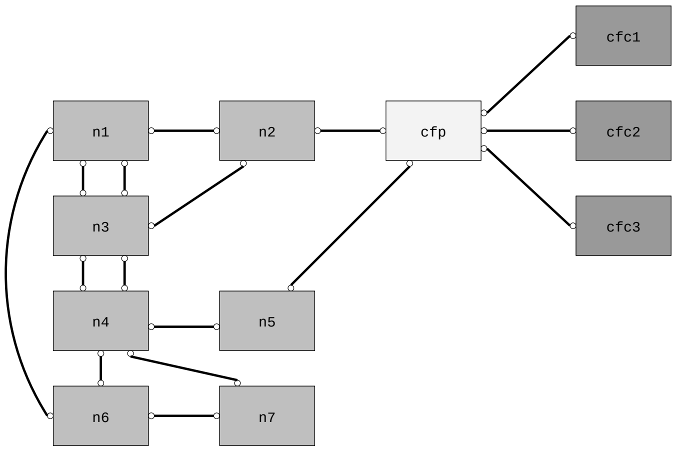


Fig. 3. A "Concrete" Architectural Description (graphical)

language. Figure 3 shows the same architecture, in graphical form. This description has been simplified by omission of detail that is irrelevant to the example, including specifications of visible behavior at component interfaces, specifications of connector behavior, and specifications of communication protocols. However, we imagine that every "box" in the description corresponds to a single software component in the implementation and that every "arrow" corresponds to a single dataflow path between software components. Thus, the description and the as-implemented archiecture are isomorphic.

There are three types of components in this architecture. The noncritical components — `n1`, `n2`, ..., `n7` — implement noncritical functions. This fact is indicated by the value of their `criticality` property, `0`. The three components `cfc1`, `cfc2`, and `cfc3` each independently determine the result of performing `critical_function_1`, identified by the requirements as a critical function. We imagine that these three components are commerical quality, but possibly fault-prone. The final component, `cfp`, actually provides `critical_function_1` to the system. If `n2` or `n5` calls for `critical_function_1` to be performed, `cfp` relays the request to `cfc1`, `cfc2`, and `cfc3`. If two of those three components return the same result to `cfp`, then that result is used by `cfp` as the value of `critical_function_1`. Note that any one of `cfc1`, `cfc2`, and `cfc3` can fail, providing no result or an incorrect result to `cfp`, and `cfp` will still provide a correct result to the system. (For simplicity of the protocol, we assume that `cfp` is a custom, adequately fault-tolerant component, and that no connectors are fault-prone.) Since each of the four components `cfc1`, `cfc2`, `cfc3`, and `cfp` contributes to the provision of critical function `critical_function_1`, each has a non-zero value of its `criticality` property. Specifically, each has the value `1`, indicating that the requirements specify that one fault must be tolerated.

Given the small number of components and the simplification via omission already performed, further abstraction is hardly required to conclude that this architecture satisfies its

```
system sys = {
  component cfp = {
    ports { p0; p1; p2; p3; p4; };
    property functions_provided = {critical_function_1};
    property criticality: int = 1;
  };
  component cfc1 = {
    ports { p; };
    property functions_computed = {critical_function_1};
    property criticality: int = 1;
  };
  component cfc2 = {
    ports { p; };
    property functions_computed = {critical_function_1};
    property criticality: int = 1;
  };
  component cfc3 = {
    ports { p; };
    property functions_computed = {critical_function_1};
    property criticality: int = 1;
  };
  component n1 = {
    ports { p0; p1; p2; p3; };
    property functions_computed = {noncritical_function_1};
    property criticality: int = 0;
  };
  component n2 = {
    ports { p0; p1; p2; };
    property functions_computed = {noncritical_function_2};
    property criticality: int = 0;
  };
  component n3 = {
    ports { p0; p1; p2; p3; p4 };
    property functions_computed = {noncritical_function_3};
    property criticality: int = 0;
  };
  component n4 = {
    ports { p0; p1; p2; p3; p4 };
    property functions_computed = {noncritical_function_4};
    property criticality: int = 0;
  };
  component n5 = {
    ports { p0; p1; };
    property functions_computed = {noncritical_function_5};
    property criticality: int = 0;
  };
  component n6 = {
    ports { p0; p1; p2; };
    property functions_computed = {noncritical_function_6};
    property criticality: int = 0;
  };
  component n7 = {
    ports { p0; p1; p2; p3; };
    property functions_computed = {noncritical_function_7};
    property criticality: int = 0;
  };

  connector k1  = { roles { r0; r1; }; };    connector k2  = { roles { r0; r1; }; };
  connector k3  = { roles { r0; r1; }; };    connector k10 = { roles { r0; r1; }; };
  connector k11 = { roles { r0; r1; }; };    connector k12 = { roles { r0; r1; }; };
  connector k13 = { roles { r0; r1; }; };    connector k14 = { roles { r0; r1; }; };
  connector k15 = { roles { r0; r1; }; };    connector k16 = { roles { r0; r1; }; };
  connector k17 = { roles { r0; r1; }; };    connector k18 = { roles { r0; r1; }; };
  connector k19 = { roles { r0; r1; }; };    connector k20 = { roles { r0; r1; }; };
  connector k21 = { roles { r0; r1; }; };    connector k22 = { roles { r0; r1; }; };

  attachments {
    k1.r0 to cfp.p2;    k1.r1 to cfc1.p;    k2.r0 to cfp.p3;    k2.r1 to cfc2.p;
    k3.r0 to cfp.p4;    k3.r1 to cfc3.p;    k10.r0 to cfp.p0;   k10.r1 to n2.p2;
    k11.r0 to n2.p1;    k11.r1 to n1.p2;    k12.r0 to n1.p1;    k12.r1 to n3.p1;
    k13.r0 to n1.p0;    k13.r1 to n3.p0;    k14.r0 to n2.p0;    k14.r1 to n3.p2;
    k15.r0 to n3.p3;    k15.r1 to n4.p1;    k16.r0 to n3.p4;    k16.r1 to n4.p0;
    k17.r0 to n1.p3;    k17.r1 to n6.p2;    k18.r0 to n4.p2;    k18.r1 to n5.p1;
    k19.r0 to n4.p4;    k19.r1 to n6.p0;    k20.r0 to n4.p3;    k20.r1 to n7.p0;
    k21.r0 to n6.p1;    k21.r1 to n7.p1;    k22.r0 to cfp.p1;   k22.r1 to n5.p0;
  };
}
```

Fig. 4. A “Concrete” Architectural Description (Acme)

```
transformation bundle_noncritical_components from
   system @s = {
     component @c1 = {
       ports {@op; @@ps1};
       property functions_computed = {@@fns1};
       property criticality: int = 0;
     };
     component @c2 = {
       ports {@ip; @@ps2};
       property functions_computed = {@@fns2};
       property criticality: int = 0;
     };
     connector @k = { roles {@ir; @or}};
     attachments {
       @k.@ir to @c1.@op;
       @k.@or to @c2.@ip;
       @@att
     };
     @@rest
   }
to
   system @s = {
     component @&(@c1, @c2) = {
       ports {@*(@_(@c1,@@ps1), @_(@c2,@@ps2))};
       property functions_computed = {@*(@@fns1, @@fns2)};
       property criticality: int = 0;
     };
     attachments {
       @@att
     };
     @@rest
   }
where
   map connected_components_to_component with {
     [first_component = @c1; first_port = @c1.@op;
          connecting_connector = @k; first_role = @k.@ir; second_role = @k.@or;
          second_component = @c2; second_port = @c2.@ip]
        to @&(@c1, @c2);
     @.(@c1,@@ps1) to @.(@&(@c1,@c2),@_(@c1,@@ps1));
     @.(@c2,@@ps2) to @.(@&(@c1,@c2),@_(@c2,@@ps2));
   }
```

Fig. 5.   An Abstraction Transformation that Preserves the Fault-Tolerance Property

fault-tolerance requirement. However, substantial further abstraction is possible, and will serve to illustrate our general method for generating maximally abstract descriptions. The basic idea is to eliminate all information in the specifcation that is irrelevant to determining whether the architecture is adequately fault tolerant. In order to do so, we repeatedly apply the two abstraction transformations. This first, shown in Figure 5, says that a connected pair of noncritical components can be collapsed to a single component. The transformation consists of two Acme specification patterns[1] and a mapping between then. Applying a transformation to a description consists of finding a match to the first pattern, and using the pattern variable values to generate a description that matches the second pattern.[2] The second transformation used in abstraction says that multiple connectors between a pair of components can be collapsed to a single connector. Clearly, neither of these bundling transformations can produce an abstract architectural description that satisfies the fault-tolerance requirement from a concrete architectural description that does not.[3]

Using these two abstraction transformations to generate a maximally abstract description is straightforward: if the first

[1]An Acme specification pattern is simply an Acme description with some elements replaced by pattern variables, which are written with an initial `@`, and some sequences of elements replaced sequence pattern variables, which are written with an initial `@@`. (Operators `@.`, `@_`, and `@&` are used to systematically rename elements to avoid potential name collisions, and the operator `@*` is used to concatenate the values of sequence pattern variables.)

[2]In general, transformations can contain additional constraints on the description that must be satisfied in order to have a match, but this feature is not required for our example.

[3]The co-preservation of fault-tolerance could be formally verified by proving that the transformations always produce an abstract architecture whose theory is faithfully interpretable in the theory of the concrete architecture [Moriconi, Qian & Riemenschneider 95; Riemenschneider 97; Riemenschneider 98].

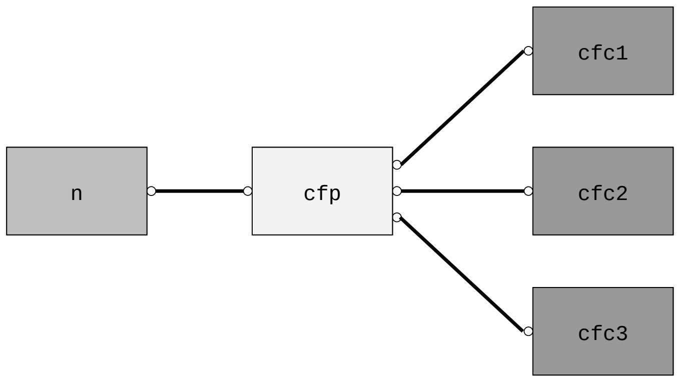


Fig. 6. A "Abstract" Architectural Description (graphical)

transformation can be applied to the most abstract description, it is, creating a new "most abstract description"; if the first cannot be applied, but the second can, it is; if neither can be applied, the abstraction process is finished. The end result is the collapse all noncritical components into a single component, critical components are left alone, and all connected components are singly connected. So, for example, the maximally abstract (for this fault tolerance property) of the architecture shown in Figure 3 is shown in Figure 6.

In this case, the analysis of the abstract architectural description is so straightforward that neither a theorem prover nor a model checker is required to determine whether the fault-tolerance requirement is satisfied: if the criticality level of some function is n, then that function must be computed by more than $2n$ components, so analysis simply consists in counting the number of components that compute each critical function.

The feasibility of automatically updating gauges is demonstrated by integrating additional components that redundantly compute a second critical function into the running system. Thus, the system's requirements, functionality, and architecture are all dynamically modified. Replaying the derivation consists applying the same abstractions to generate a new maximally-abstract fault-tolerance model of the system. ("The same abstractions" means that the same abstraction transformations are applied, in the same order, using pattern variable bindings that are almost the same — the sequence pattern variable bindings may differ — as in the original derivation. If the attempt to replay breaks down, the transformation selection strategy can often be used to "fill in the gaps".[4]) The analysis of the abstract model determines that there are now two critical functions rather than one, and generates a second fault-tolerance gauge whose reading shows whether the implementation of the second critical function is adequately redundant.

[4]Discovering the extent to which this approach to replay allows derivations to be automatically updated is one of our principal research foci. Another principal focus is determining when and how analyses performed by theorem provers and model checkers can be automatically and efficiently updated. Both derivation and analysis must be "robust" for automatic updating of dependability gauge readings.

VI. Related Work

Transformational implementation was an outgrowth of earlier work on program synthesis. The basic idea is to formalize the process of refining a high-level program specification into executable code. Many of the seminal papers in the field have appeared in anthologies [Section V of Agresti 86, Section III of Rich & Waters 86, Section 5 of Lowry & McCartney 91]. Experience showed that refinement was a knowledge-intensive process, and the key to success was to focus on a relatively narrow domain. Software architecture is one such domain. At SRI, we have worked on the formalizing the process of architecture refinement since 1992 [Moriconi, Qian & Riemenschneider 95; Moriconi, *et al.* 97; Moriconi & Riemenschneider 97; Riemenschneider 97; Riemenschneider 98; Riemenschneider 99; Gilham, Riemenschneider & Stavridou 99; Herbert, *et al.* 99]. Recently, we have focused on adapting our technology to component-based systems [Riemenschneider & Stavridou 99]. Although some other researchers have investigated the notion of mappings between architectural descriptions at different levels of abstraction [Luckham, *et al.* 85], none, to the best of our knowledge, has attempted to formalize the process of generating the mappings by using transformations that are "correctness preserving". Conversely, most researchers investigating the notion of correct refinement have not focused on refinement of architecture. There are a few exceptions (e.g., [Broy 92]). Of particular note is the work of Philipps and Rumpe at T. U. Munich [Philipps & Rumpe 97], who explicitly address the problem of correct architecture refinement and that of Saridakis and Issarny of IRISA [Saridakis & Issarny 99] on developing dependable architectures by refinement. Neither employs a transformational framework, however.

Abstract interpretation is the general framework for the definition of abstractions of programs. It consists of a mapping between a concrete and an abstract domain that sends sets of concrete states to single abstract states, together with a mapping from the basic operations or functions of the concrete system to functions of the abstract system. While abstract interpretation is the basis for static analysis techniques used in compilers, it is not widely used for dependability analysis. It is in fact extremely difficult to construct useful and accurate abstractions that automatically preserve the desired dependability properties. Abstract models are usually provided manually, and theorem proving is used to check that the abstraction mapping preserves the properties. Once the preservation property is established using theorem proving, the abstract model is analyzed by model checking. Recently [Graf & Saïdi 97; Saïdi & Shankar 99], novel techniques for automatic Boolean abstraction have been developed by SRI. These techniques enable verification of system temporal properties of infinite state systems without manual construction of an abstraction.

Since the abstraction process introduces loss of information by collapsing concrete states into a single abstract state, false negative results may emerge. For instance, a model checker may exhibit an error trace that corresponds to an execution of the abstract program that violates the dependability properties. However, this error trace may not correspond to an execution trace in the concrete program. This situation indicates that the abstraction is too coarse. That is, too many details were abstracted away, and the abstraction needs to be refined. Techniques have been developed recently at SRI [Saïdi 99] to use the error trace to automatically refine the abstraction. The verification methodology — abstraction followed by successive refinements based on the results of model checking — was successfully used to prove safety properties of several systems, including a data link protocol used by Philips Corporation in one of its commercial products. The original proof of the protocol required two months of work and was entirely done using a theorem prover. A Boolean abstraction of the protocol can be automatically generated using the predicates appearing in the description of the protocol in about a hundred seconds with SRI's PVS theorem prover. The abstract protocol is then analyzed in a few seconds to check that all the safety properties hold.

VII. Conclusion

Dependability gauges provide a technology to monitor evolving dependability properties of dynamically evolving systems. Our approach to building dependability gauages applies proven technology for design time dependability analysis at runtime. The principal innovation consists of focussing abstraction rather than refinement, and on automatic updating of abstractions and analyses developed at design time after making small, "well structured" changes to architectural requirements and the system architecture. Our emphasis in the near future will be to develop a suite of abstraction transformations capable of generating a wide range of dependability gauges, and on developing and experimenting with technologies to make abstraction and analysis more robust.

Our dependability gauge technology is complementary to the more fine-grained runtime analysis that can be performed by monitoring events at component interfaces and within connectors. The fault-tolerance gauges in our example provide an excellent example of the potential synergy. Runtime event monitoring of components' interface behavior can detect some instances of component failure. If failure of a component computing a critical function is observed, and the fault-tolerance gauge for that function shows that only a single component failure can be tolerated, the system, although functioning correctly, is on the verge of failure, and immediate corrective action may be needed. If, on the other hand, the fault-tolerance gauge shows that multiple component failures can be tolerated, the need for corrective action is less urgent. Thus, the combination of component runtime behavior gauges and dependability gauges provides valuable system status information that cannot be obtained with either technology alone.

References

[1] W. W. Agresti, editor, *Tutorial: New Paradigms for Software Development*, IEEE Computer Society, 1986.

[2] M. Broy, "Compositional Refinement of Interactive Systems", Report Number 89, Digital Systems Research Center, Palo Alto, CA, July, 1992.

[3] F. Gilham, R. A. Riemenschneider, and V. Stavridou, "Secure Interoperability of Secure Distributed Databases: An Architecture Verification Case Study", *FM '99, World Congress on Formal Methods*, Toulouse, France, September 20-24, 1999.

[4] S. Graf and H. Saïdi, "Construction of Abstract State Graphs Using PVS", *Proceedings of the 9th International Conference on Computer-Aided Verification, CAV '97*, Haifa, Israel, 1997.

[5] J. Herbert, B. Dutertre, R. A. Riemenschneider, and V. Stavridou, "A Formalization of Software Architecture", *FM '99, World Congress on Formal Methods*, Toulouse, France, September 20-24, 1999.

[6] M. R. Lowry and R. D. McCartney, editors, *Automating Software Design*, AAAI Press/MIT Press, 1991.

[7] D. C. Luckham, L. M. Augustin, J. J. Kenney, J. Vera, D. Bryan, and W. Mann, "Specification and Analysis of System Architecture using Rapide", *IEEE Transactions on Software Engineering*, vol. 21, no. 4, April, 1995.

[8] M. Moriconi, X. Qian, and R. A. Riemenschneider, "Correct Architecture Refinement", *IEEE Transactions on Software Engineering*, vol. 21, no. 4, April, 1995.

[9] M. Moriconi, X. Qian, R. A. Riemenschneider, and L. Gong, "Secure Software Architectures", *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, Oakland, CA, May, 1997.

[10] M. Moriconi and R. A. Riemenschneider, "Introduction to SADL 1.0: A language for specifying software architecture hierarchies", Technical Report SRI-CSL-97-01, Computer Science Laboratory, SRI International, Menlo Park, CA, March, 1997.

[11] J. Philipps and B. Rumpe, "Refinement of Information Flow Architectures", *Proceedings of the First IEEE International Conference on Formal Engineering Methods, ICFEM '97*, November, 1997.

[12] C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[13] R. A. Riemenschneider, "A Simplified Method for Establishing the Correctness of Architectural Refinements", Working Paper DSA-97-02, Dependable System Archiecture Group, Computer Science Laboratory, SRI International, Menlo Park, CA, November, 1997. Available at `http://www.sdl.sri.com/dsa/publis.html`.

[14] R. A. Riemenschneider, "Correct Transformation Rules for Incremental Development of Architecture Hierarchies", Working Paper DSA-98-01, Dependable System Archiecture Group, Computer Science Laboratory, SRI International, Menlo Park, CA, February, 1998. Available at `http://www.sdl.sri.com/dsa/publis.html`.

[15] R. A. Riemenschneider. "Checking the Correctness of Architectural Transformation Steps via Proof-Carrying Architectures", in *Software Architecture*, ed. P. Donahoe, Kluwer, 1999.

[16] R. A. Riemenschneider and V. Stavridou. "The Role of Architecture Description Languages in Component-Based Development: The SRI Perspective", *1999 International Workshop on Component-Based Software Engineering*, Los Angeles, CA, May 17-18, 1999.

[17] H. Saïdi and N. Shankar, "Abstract and Model-Check While You Prove", *Proceedings of the 11th International Conference on Computer-Aided Verification, CAV '99*, Trento, Italy, 1999.

[18] H. Saïdi, "Modular and Incremental Analysis of Concurrent Software Systems", *Proceedings of the 14th IEEE International Conference on Automated Software Engineering, ASE '99*, Cocoa Beach, FL, 1999.

[19] T. Saridakis and V. Issarny, "Developing Dependable Systems Using Software Architecture", in *Software Architecture*, ed. P. Donahoe, Kluwer, 1999.