24. End-to-End (E2E) Testing and Evaluation of High-Assurance Systems

U. S. Department of Defense (DoD) end-to-end (E2E) testing and evaluation (T&E) technology for high-assurance systems has evolved from specification and analysis of thin threads, through system scenarios, to scenario-driven system engineering including reliability, security, and safety assurance, as well as dynamic verification and validation. Currently, E2E T&E technology is entering its fourth generation and being applied to the development and verification of systems in service-oriented architectures (SOA) and web services (WS). The technology includes a series of techniques, including automated generation of thin threads from system scenarios; automated dependency analysis; completeness and consistency analysis based on condition–event pairs in the system specification; automated test-case generation based on verification patterns; test-case generation based on the topological structure of Boolean expressions; automated code generation for system execution as well as for simulation, automated reliability assurance based on the system design structure, dynamic policy specification, analysis, enforcement and simulation; automated state-model generation; automated sequence-diagram generation; model checking on system specifications; and model checking based on test-case generation. E2E T&E technology has been successfully applied to several DoD command-and-control applications as well civilian projects.

The Department of Defense (DoD) end-to-end testing and evaluation (E2E T&E) project started in 1999 when the DoD was involved in the largest testing project ever, i.e, year 2000 (Y2K) testing. During Y2K testing, it was discovered that, even though DoD had many testing guidelines, most of them only addressed unit testing, and few were available for integration testing, but Y2K testing involved mainly integration testing, and thus needed E2E T&E guidelines.

Even though many techniques, such as inspection and program verification, are available for evaluating system reliability and quality, testing was and is the primary means for reliability and quality assurance. Furthermore, in practice, integration testing is often the most time-consuming and expensive part of testing. It is common to find software development projects with 50–70% of effort on testing, and 50–70% of the testing effort on integration testing. A review of the literature on integration testing shows that most integration testing techniques are either a methodology, such as incremental integration, top-down, and bottom-up integration [24.1], or are based on specific language or design structures of the program under test [24.2–5]. These techniques are useful, but are applicable to software written using the related techniques only. For example, an integration testing technique for an object-oriented (OO) program using Java may not be applicable to the testing of a legacy program using the common business-oriented language (COBOL). It may not be applicable to a C++ program because Java has no pointers but C++ does.

Due to these considerations, DoD initiated a project on E2E T&E in 1999 [24.6], intended to verify the interconnected subsystems as well as the integrated system. E2E T&E is different from module testing where the focus is on individual modules and is similar to, yet different from, integration testing where the focus is on the interactions among subsets of modules. Since 1999, E2E T&E has evolved from thin-thread specification and analysis, to scenario specification and analysis, and to scenario-driven system engineering (SDSE), and from SDSE to testing and verification of web services (WS) in a service-oriented architecture (SOA).

This paper is organized as following. Section 24.1 covers the history and evolution of E2E T&E and scenario specification. Section 24.2 presents an overview of the third and fourth generations of E2E T&E techniques. Section 24.3 elaborates static analyses, including model checking, completeness and consistency (C&C) analyses, and test-case generation. Section 24.4 presents automated test execution by distributed agents and how simulation of concurrent scenarios can be executed. Section 24.5 discusses policy specification and enforcement, which can be used to enforce safety and security policies, as well as dynamic verification and validation. Section 24.6 presents the reliability model for dynamic reliability assurance. Section 24.7 outlines the application of E2E T&E in SOA. Finally, Sect. 24.8 concludes this paper.

24.1 History and Evolution of E2E Testing and Evaluation

This section briefly describes the evolutionary development of the new generations of E2E T&E. Table 24.1 depicts the four generations of E2E T&E, their application periods, and the signature techniques in each generation.

24.1.1 Thin-Thread Specification and Analysis – the First Generation

The genesis of the DoD E2E T&E is thin-thread specification and analysis. This is based on the lesson learned from DoD Y2K testing. At that time, it was discovered that the DoD did not have an integration testing guideline that could be used for a variety of applications written in a variety of programming languages. Most existing integration techniques are either mainly of high-level concepts (such as those that used an incremental manner to perform integration testing) or are applicable to specific design structures or programming languages only, e.g., an integration testing techniques for object-oriented (OO) programs. Thus, there is an immediate need for an integration testing guideline that can be used by a majority of DoD organizations and services.

While no such DoD integration testing guidelines are available, it was discovered that most DoD organizations used the concept of thin threads to perform integration testing. A thin thread is essentially an execution sequence that connects multiple systems during system exercise and execution, and most organizations reported their Y2K testing effort in terms of the number of thin threads successfully executed and tested. In other words, thin threads were successfully used as the

Table 24.1 Evolution of E2E T&E techniques

Generations	Application period	Signature techniques
First	1999–2002	Thin thread specification and analysis techniques
Second	2001–2003	Scenario specification, analysis, and pattern verification techniques
Third	2003–present	Scenario specification and analysis, Scenario-driven system engineering, including reliability, security, and risk analysis; modeling and simulation
Fourth	2004–present	Scenario specification and analysis, and scenario-driven system engineering in service-oriented architecture with dynamic composition and recomposition

principal technique for Y2K integration testing. Integration testing based on thin threads has many advantages including:

- Thin threads are independent of any application;
- Thin threads are independent of any programming languages;
- Thin threads are also independent of any specific design structure;
- Thin threads can be used early during system development and late during system integration testing; and
- Thin threads can be easily understood by a vast number of DoD engineers. Thus, a DoD integration testing based on thin threads becomes a viable candidate for an integration guideline.

However close examination of DoD Y2K testing effort also revealed some important weakness of thin threads:

- Most thin threads were specified without a consistent format or using a localized format. In other words, different groups used different formats to specify thin threads;
- Most thin threads were developed manually and placed in an Excel file and thus were rather expensive to develop and maintain;
- The number of thin threads needed for successful integration testing was not known and thus some organizations used an extensive number of thin threads (such as thousands of thin threads) while some used only few (such as four to five) for a large application;
- The quality of thin threads was not easy to determine as they were developed manually and verified manually.

Thus, the first step of DoD E2E T&E focuses on the following issues:

1. The development of a consistent format for specifying thin threads. Because many thin threads share certain commonality with other thin threads, the DoD E2E guideline also suggests the organization of thin threads into a hierarchical thin-thread tree with related thin threads grouped together as a sub-tree in the thin-thread tree;
2. The development of a tool so that thin threads can be analyzed to ensure that these thin threads meet the minimum requirements;
3. The development of a guideline to determine the number of thin threads needed for an application; specifically, assurance-based testing (ABT) was developed to determine the number of thin threads needed for a certain system quality.

This first-generation DoD E2E T&E also assumes that each individual, participating system has been tested before they are subject to the DoD integration testing.

Several related techniques to thin threads have also been developed, including functional regression testing [24.7], automated dependency recognition and analysis, risk analysis, and test coverage based on specification of thin threads. Three versions of the DoD E2E tools have been developed and experimented with in the period 1999–2002. This experimentation showed that, once the thin-thread tree is specified, it is straightforward to develop test cases to run the integrated system.

24.1.2 Scenario Specification and Analysis – the Second Generation

During 2001–2002, experimentation of the DoD thin-thread tools revealed several serious shortcoming of thin threads:

- The number of thin threads needed is often too large to be manually developed even if an automated support tool is available; and
- Many thin threads differ only slightly from each other as they addressed the same similarity features and functionality of the application.

These shortcomings are due to the fact that each thin thread represents a specific execution sequence while a typical DoD application may have numerous execution

sequences. Thus, it is expensive and time-consuming to specify these thin threads even with automated tool support. To address these problems, it was discovered that it is possible to add control constructs such as *if-then-else* and *while* into thin threads. However, adding these constructs will change the meaning of thin threads because such a modified thread no longer represents one execution sequence, but multiple execution sequences. In other words, the modified thread is no longer a thin thread as defined by the DoD Y2K testing project, and the modified thread is called a scenario, for lack of an alternative, better names.

DoD E2E T&E is changed from the specification and analysis of thin threads to the specification and analysis of system scenarios; furthermore, techniques are developed so that thin threads will be automatically generated once system scenarios are specified. The fact that thin threads can be automatically generated from E2E scenarios makes them different from unified modeling language (UML) use cases. While UML use cases also describe system scenario from an external point of view, a use case does not need to be able to generate thin threads for testing, furthermore the E2E system scenarios can be used early in the development life cycle as well as late for integration testing and regression testing.

The original DoD E2E T&E techniques, such as automated dependency recognition and analysis, test coverage, risk analysis, functional regression testing, are modified so that they are applicable to the E2E system scenarios. Furthermore, several versions of tools were developed to support scenario specification and analysis.

Experiments with second-generation DoD E2E T&E were carried out on several projects, including a testing project for a high-availability communication processor. The requirements of a sample telecommunication processor were first translated into system scenarios, then the tool automatically generate a large number of thin threads from these scenarios, and finally an engineer developed test cases based on the thin threads generated. It was discovered that translating the original requirements into system scenarios is much easier than specifying the thin threads from the same requirements, and generating test cases from the thin threads generated is straightforward. The engineers involved in these experimentation also expressed the advantage of this approach over their current approach. It was much more difficult to develop test cases from the system requirements using their current approach, and the DoD E2E approach is much more structural and rigorous while saving them time and effort in developing test cases. In conclusion, second-generation DoD E2E T&E achieved its original goal of assisting test engineers to perform integration testing efficiently and effectively.

Several other new techniques were developed and discovered:

1. The system scenarios can be specified formally and subjected to a variety of formal analyses, not just the dependency analysis and risk analysis developed in the first generation of the E2E tool.
2. Systems often exhibit patterns in their behavior and these patterns can be rather useful for automated test-script generation.
3. As the developers of DoD E2E T&E always suspected, E2E techniques can also be useful in design and analysis rather than for integration testing only. This was confirmed in early 2003, and after hearing the briefing of the E2E T&E, a DoD organization started using the E2E T&E techniques for specifying and analyzing its command-and-control system.

Formalized scenario specification

The system scenario in DoD E2E T&E can be formalized in two ways: the first concerns the elements in the scenario, while the second concerns the process aspect of the scenario. The first aspect is formalized using the the actor, condition, data, action, timing, and event (ACDATE) model [24.8]. Using the ACDATE model, the specification of the software under development is described by its five model elements and their relationships.

- An **actor** is a model element that represents a system or its component with a clear boundary that interacts with other actors.
- A **condition** is a predicate on data used to determine the course of a process taken by actors. Conditions can be preconditions and post-conditions representing external and internal conditions and situations. Internal conditions represent the states of all system objects of interest, and external conditions can be network and database connections
- A **data** is an information carrier that represents the internal status of actors.
- An **action** is a model element that represents an operational process to change the internal status of an actor. Actions are performed when the preconditions are satisfied and events occur. Typically an action is
  a brief atomic computation such as
  – Assignment: sets the value of a variable,

  - Call: calls an operation on a target object,
  - Create: creates a new object,
  - Destroy: destroys an object,
  - Return: returns a value to a caller.
  - Send: generates an event, outgoing data,
  - Terminate: self-destruction of the owning object.
- The **timing** is an attribute of an actor, data, condition, event and action or behavioral model elements that describe their static or dynamic time information.
- An **event** is a model element that represents an observable occurrence with no time duration. Events can be internal and external occurrences that impact on, or are generated by, system objects such as incoming data (inputs), external action, and internal method call/message.

Once the specification of the software under development is represented by these five components and their relationships, the execution steps can be constructed using control constructs such as *if-then-else* and *while-do*, as shown in the following example. Figure 24.1 illustrates a simple scenario: "when both the driver and passenger door are locked, if remote controller is pressed for unlock, then the driver door will be opened", in the design of a car alarm system. In this scenario, five actors, one condition, one data, and one action are used.

Once the system scenario are specified, model checking, test-case generation, automated code generation, policy-enforcement-based dynamic testing, and simulation can be performed.

An important attribute is that scenarios can be specified in a hierarchical manner. The tester can first specify system scenarios at the highest level of abstraction. Once obtained, scenarios can be decomposed to show low-level details. This process can continue until scenarios are detailed enough for the T&E purpose. Furthermore, scenarios can be organized in a scenario tree where a group of related scenarios form a high-level scenario group [24.9, 10]. This feature is useful for testing an system of systems (SoS) because it often has subcomponents that interact with each other, and some of these components are legacy systems while others may be new systems that have just been introduced. Organizing system scenarios in a hierarchical manner facilitates test reuse and matches the hierarchical structure of the SoS.

Pattern Analysis

Even though a system may have hundreds of thousand scenarios, it may have only a few scenario patterns. For example, a commercial defibrillator has hundreds of thousand of scenarios, however, most (95%) of these scenarios can be classified into just ***eight*** scenario patterns [24.11]:

- Basic pattern (40%),
- Key-event-driven pattern (15%),
- Timed key-event pattern (5%),
- Key-event time-sliced pattern (7%),
- Command–response pattern (8%),
- Look-back pattern (6%),
- Mode-switch patten (8%), and
- Interleaving pattern (6%).

This provides an excellent opportunity for rapid verification because scenarios that belong to the same pattern can be verified using the same mechanism, except perhaps with different parameters such as timing and state information. This can save significant time and effort for implementation of test scripts.

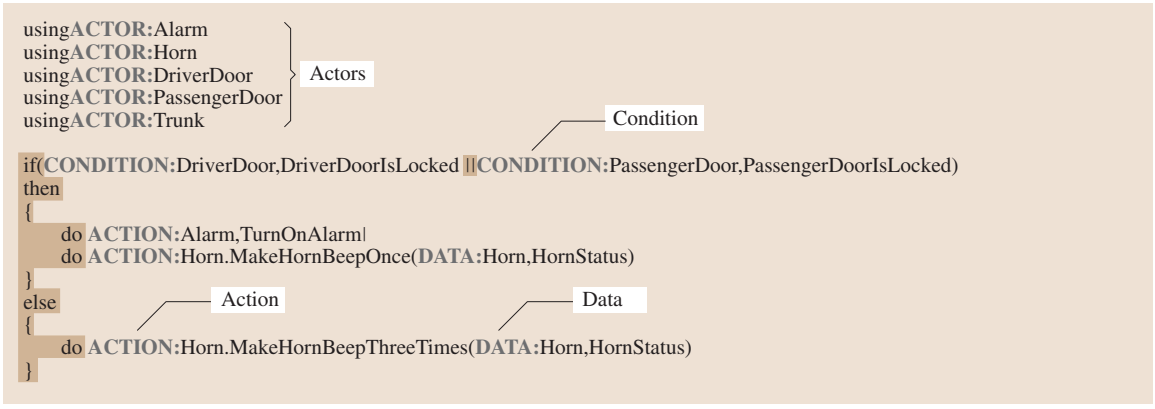

Fig. 24.1 A sample scenario in the ACDATE language. Once the system scenarios are specified, model checking, test-case generation, automated code generation, policy-enforcement-based dynamic testing, and simulation can be performed

For example, suppose a system has 7000 scenarios, and if 15% of these scenarios belong to a specific pattern, these 1050 (7000×0.15) scenarios can be tested using the same verification software with individualized parameters. Thus, the productivity gain can be significant and industrial application of this approach showed that 25–90% effort reduction is possible [24.12].

Another significant advantage of this approach is the size reduction achieved using this approach. Industrial applications and experiments have indicated that the average length of code for test scripts reduced from 1380 lines of code (LOC) per scenario to 143 LOC per scenario using this approach, corresponding to a size reduction of 89.6%. If we assume that an expert test engineer can develop 1000 LOC of test script each week, the effort reduction achieved by using this approach is significant.

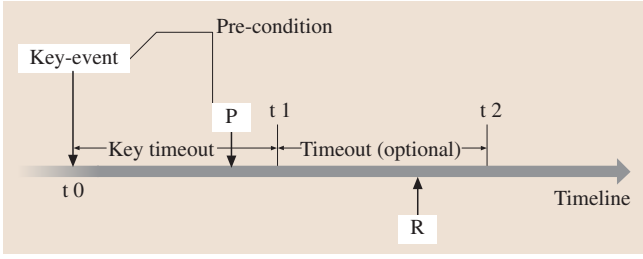

Fig. 24.2 Timed key-event-driven requirement pattern

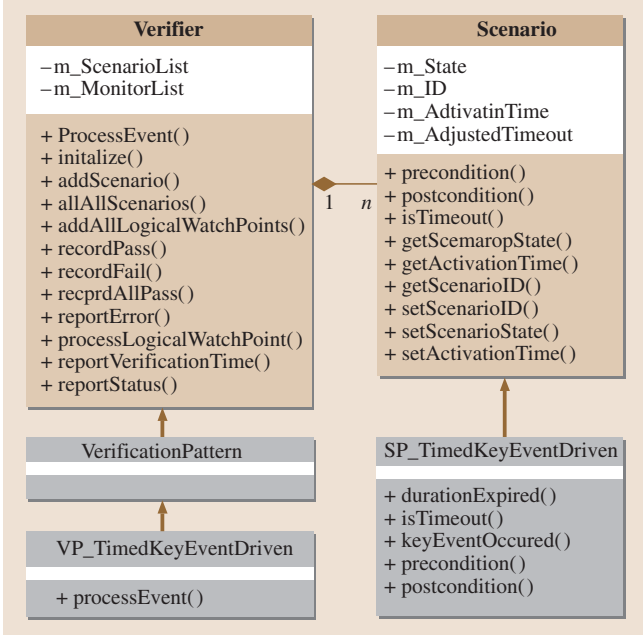

Fig. 24.3 Class diagram of the timed key-event-driven requirement pattern

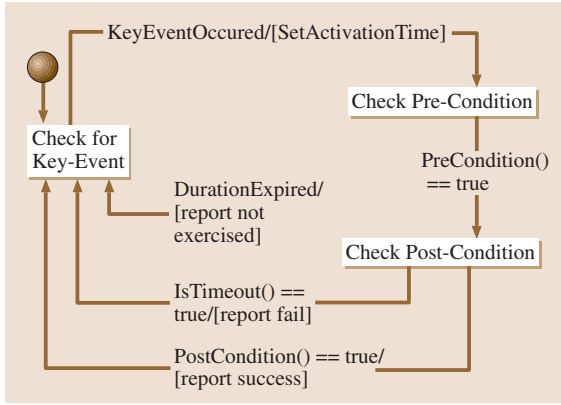

Fig. 24.4 Timed key-event-driven verification pattern

The following illustrate this concept for a timed key-event pattern.

Figure 24.2 shows a timed key-event-driven scenario pattern that includes two timing constraints for three events:

- Within the duration from t0 to t1 and after the key event, if event P occurs, then event R is expected to occur before t2.

As a typical example in an implantable defibrillator, when the device detects a heart problem, the capacity must be charged before it can apply a therapy to the patience, and this scenario shows three events (detection, capacity charged, and therapy applied), and the timing constraints between these three events.

Timed key-event verification patterns.
Name. Timed key-event-driven verification pattern
Description. The timed key-event-driven verification pattern is used to verify requirements that can be represented using the timed key-event-driven scenario pattern shown in Fig. 24.3. It provides an interface to decide if the duration has expired.
Verification state machine. Unlike the basic verification pattern, which starts checking the pre-condition right away, the verification process here checks the pre-condition within the *duration* after the key event occurs. The verifier can report "*not exercised*" if it failed to verify the pre-condition within this duration. Figure 24.4 shows an example of the timed key-event-driven verification pattern.

24.1.3 Scenario-Driven System Engineering – the Third Generation

Once system scenarios are formalized and experimented, DoD realized another need, i. e., can DoD E2E technology be useful for system engineering and system development? Traditional system engineering focuses on the following aspects:

- Reliability analysis;
- Safety analysis;
- Security analysis;
- Simulation, including distributed simulation and code generation;
- Verification and validation (V&V).

The DoD E2E already focused on V&V, and thus the quest is to extend the E2E technology to address the other aspects of system engineering. The rest of the paper will focus on

- Distributed simulation and code generation will be discussed in Sect. 24.4;
- Safety analysis: traditional safety analysis includes event analysis, event sequence and fault-tree analysis, while modern safety analysis includes static model checking and dynamic simulation analysis using executable policies. Model checking will be discussed in Sect. 24.1 and policy specification and enforcement will be discussed in Sect. 24.6;
- The DoD E2E security analysis is based on specification security policies and uses the simulation to evaluate the system vulnerability by verifying the security policies at runtime. These will be discussed in Sects. 24.4 and 24.6.
- Reliability analysis: it turns out that system scenarios are useful for both static and dynamical reliability analyses, and this will covered in Sect. 24.6.

24.1.4 E2E on Service-Oriented Architecture – the Fourth Generation

Currently, E2E T&E technology is being applied to the emerging SOA and web services (WS) platforms where more dynamic features are required, including dynamic composition, recomposition, configuration, reconfiguration, V&V, reliability assurance, ranking of WS, and methodologies that assess the WS. The fourth generation of E2E T&E has the same basic techniques but is implemented on a different software architecture. The basic techniques in the third and fourth generations of E2E T&E will be discussed in Sects. 24.3–24.7 and the SOA-specific techniques will be presented in Sect. 24.8.

24.2 Overview of the Third and Fourth Generations of the E2E T&E

This section outlines the major components of the basic techniques in the third and fourth generations of the E2E T&E. As shown in Fig. 24.5, the development process starts from the user requirements, which

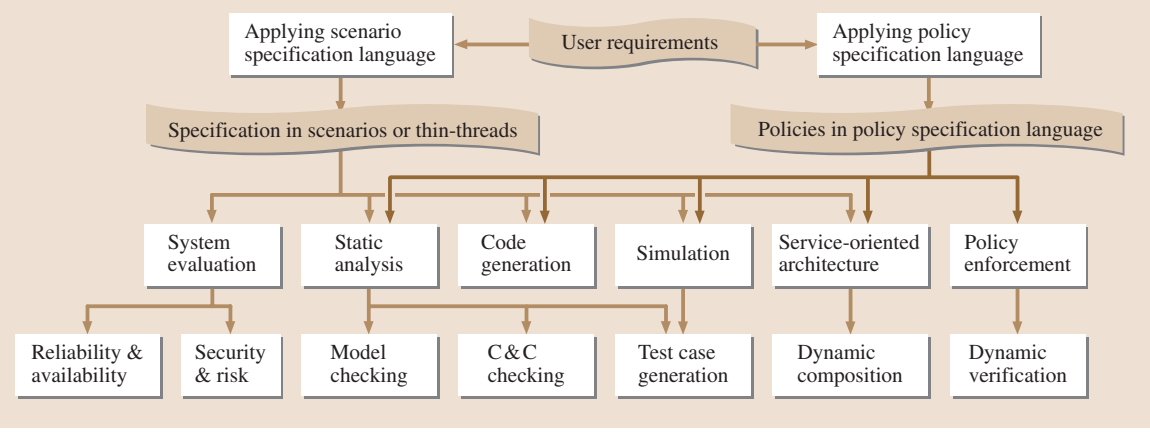

Fig. 24.5 The overall development and E2E T&E

is then formalized into the specification in the ACDATE scenario language, consisting of a sequence of actors, conditions, data, actions, timings, and events. Because scenarios are developed directly from the requirements, they are independent of any programming languages, design techniques, or development processes, such as the waterfall model, or agile development processes such as extreme programming [24.13].

Based on the scenario specification, static analysis, automated code generation, simulation, and system evaluation can be performed. E2E T&E also supports the service-oriented architecture (SOA) where software components are defined by standard interfaces that allow dynamic composition of new services based on existing services. On the other hand, the policies can be extracted from the user requirements and presented in a policy specification language. A policy-based system can be developed and policies are used to verify the behavior of the system dynamically. Static analysis, simulation, and code generation can be applied on policy specification.

Static analysis. Once the system is specified in scenarios, various analysis techniques [24.9, 14, 15] can be used to statically analyze the specification, for example, model checking and C&C. These analyses help the designer to make informed and intelligent decision in the requirement and design phases of project development [24.7, 13, 15]. In other words, the E2E T&E tool can be used early in the life cycle, during, and throughout the rest of the life cycle, including during operation and maintenance. Several methods can be applied to perform model checking, for example, Berkeley lazy abstraction software verification tool (BLAST) [24.16] developed at the University of California at Berkeley and C&C analysis [24.11]. In the process of model checking, both positive and negative test cases can be generated [24.17]. The positive test cases are used to test if the system generates correct output for valid inputs, while the negative test cases are used to test if the system does not generate an undesired output. An undesired output is one that could cause an undetected error or an unacceptable consequence.

The E2E process also supports rapid test-case generation by classifying system scenarios into patterns, where each pattern has a corresponding test case that can be parameterized to test all the system scenarios belong to the pattern. This approach promotes test-case reusability and reduces the cost of test-script generation significantly [24.18]. This approach has been used successfully to test commercial real-time safety-critical embedded medical devices such as pacemakers and defibrillators.

Code generation. Once the model is verified, the automated code-generation tool can be applied to generate the executable directly from the scenario specification. Code generation can also be performed in the simulation framework.

Simulation. Simulation is a practical way to prove the design idea and assess the performance of complex systems dynamically [24.19]. The traditional simulation is done in a specify-and-code or model-and-code manner, which means that, in the simulation process, engineers first construct the target system model and then develop the simulation code to run the simulation. This approach is expensive and inflexible. The automated code generation in E2E T&E environment can perform model-and-run or specify-and-run simulation. In other words, once the scenario model is constructed, no additional simulation coding effort is needed to run the simulation. Even the real target system's code can be automatically generated from the system model with little or no human involvement, because the same automated code-generation tool is applied to generate the real system code and simulation code. In E2E simulation, once scenarios are available, the system is executed in a simulation environment by tracing the conditions and actions in the scenarios. Simulation can be used together with other analyses to prove the ideas and features of the system design. For example, simulation can be used together with timing analysis to determine if the system satisfies the timing requirements. Furthermore, multiple scenarios can be simulated at the same time to determine the interaction of these scenarios. The E2E tool supports distributed test execution by providing the architecture with a test master and test agents. The test master is responsible for managing test scenarios and test scripts, and sending test commands to test agents for remote execution. Test agents are responsible for sending test commands to the system under test for test execution, collecting and data analysis, and for reporting test results to the test master.

Policy enforcement and dynamic verification. A policy-based system allows the requirements and the specification to be modified dynamically. The typical application of a policy-based system is in dynamic safety and security enforcement where the safety and security of a system can change from time to time. A policy-based system is also useful when the system is dealing

with a dynamically changing environment or the functional requirements can change from time to time. For example, the initial system has been programmed for an application temperature range of 0–100 degrees. If the range is later extended to 0–150 degrees, a policy-based implementation does not need to modify the program code and regenerate the executable code. Only the policy data need be updated and reloaded. Policy enforcement can be applied as a dynamic V&V method. In fact, many functional requirements of a system can be extracted as policy requirements. For example, array index range checking, probability value range checking, and execution-order checking can be written as policies. As a result, policy enforcement can dynamically check the validity of computing. Policy enforcement is particularly useful in detecting bugs that are difficult to catch during unit testing and those with complicated interactions due to concurrent threads and processes in simulation.

System evaluation. E2E T&E supports both static and dynamic analyses of reliability, availability, security, risk, timing, usage, dependency, and the effectiveness of test cases. The evaluation results can be applied immediately to guide subsequent testing. For example, according to the number of faults each test case detects, the effectiveness of the test cases can be ranked dynamically. In subsequent testing, the more effective test cases will be applied first in testing. The reliability evaluation results can be applied in selecting components that need to be replaced dynamically.

Service-oriented architecture. The service-oriented architecture (SOA) considers a software system consisting of a collection of loosely coupled services. These services can make use of each other services to achieve their own desired goals and end results. Simple services can cooperate in this way to form a complex service. Technically, a service is the interface between the producer and the consumer. From the producer's point of view, a service is a function that is well defined, self-contained, and does not depend on the context or state of other functions. In this sense a service is often referred to as a service agent. The services can be newly developed applications or just wrapped around existing legacy software to give them new interfaces. From the consumer's point of view, a service is a unit of work done by a service provider to achieve desired end results for a consumer. The next generation of E2E T&E will deal with SOA and composition and recomposition, dynamic configuration and reconfiguration of software systems. Initial investigations have been performed [24.14, 17, 20, 21].

24.3 Static Analyses

To ensure the correctness of the specification, static analysis will be performed, including model checking and C&C analysis. Test cases can be generated in the process of static analysis.

24.3.1 Model Checking

Model checking has been proposed recently to facilitate software testing following the idea that model checking verifies the model while testing validates the correspondence between the model and the system. One of the most promising approaches was proposed at the University of California at Berkeley using BLAST [24.16]. The BLAST model checker is capable of checking safety temporal properties, predicate-bound properties (in a form that asserts that, at a location l, a predicate p is true or false), and identify dead code. BLAST abstracts each execution path as a set of predicates (or conditions) and then these predicates are used to generate test cases to verify programs. This approach is attractive because it deals with code directly rather than the state model used in traditional model checking [24.22]. Thus, the BLAST approach is better suited for software verification than traditional model checking. However, BLAST does not handle currency and its test-case generation is targeted mainly on the positive aspects of testing. Negative aspects such as near misses are not handled. In our E2E T&E framework, many scenarios may be active at the same time, and it is necessary to verify that concurrent execution of these scenarios will not cause the system to deviate from its intended behavior. We extend the BLAST approach to suit the scenario specification in three ways: (1) instead of using the source code to drive model checking, we use our scenario modeling language for model checking. The control-flow automata used by BLAST resembles the workflow model derived by the control constructs in the scenario language; (2) we rely on the conditional or unconditional output, effect, and precondition in each thin thread to construct their essential inner control logic; and (3) we enhance

BLAST to handle concurrent executions of processes in the ACDATE language [24.17].

24.3.2 Completeness and Consistency Analysis

Software requirements are often incomplete, inconsistent, and ambiguous. The specification based on the requirements may have inherited the faults. Faults introduced in this stage of development have been shown to be difficult and more expensive to correct than faults introduced later in the life cycle. C&C analysis on specification aims to eliminate requirement- and specification-related faults. As shown in Fig. 24.6, the process starts from converting user requirements into a ACDATE scenario specification; extracting condition and event (CE) combinations from the specification; performing a completeness analysis to identify all the missing CE combinations; performing consistency analysis to check if the CE combinations are consistent with each other; and identifying the set of scenarios that need to be modified to make the system reliable and robust. More specifically, these steps are explained as follows.

1. Derive system scenarios from the system requirements: formalize system scenarios using the ACDATE model, which includes elements (actor, condition, data, action, timing and event) and the relations among them.
2. Parse each scenario and extract the combinations of conditions and events: from the ACDATE model, the CE combinations are automatically extracted. The CE combinations are partitioned into independent components where each component does not interact or related to the others. Two scenarios are independent of each other if there is no way for them to interact or influence each other. For example, two scenarios that share a common condition are considered related, and the related relationship is transitive. By exhaustively examining the transitive relationships, one can determine if two scenarios are independent of each other.
3. Perform C&C analysis on CE combinations: once the CE combinations are obtained in step 2, consistency analysis on CE combinations is performed and completeness analysis on CE combinations is then performed to identify those missing CE combinations.
4. Construct patching scenarios to eliminate those missing CE combinations. Using an Karnaugh-map analysis, we can aggregate a large number of missing CE combinations into a smaller set of equivalent CE combinations. From these CE combinations, we can develop patching scenarios to cover the missing condition.
5. Classify each patching scenario into one of the three categories: (1) incorporate it as a functional scenario; (2) treat it as an exception with an exception handling; or (3) consider it as a do not care item, based on the nature of the application. In the first case, the covering scenario is indeed an intended behavior but missed in the specification, in the second case the covering scenario is not intended and should be masked out in the specification.
6. Amend the scenario specification using the C&C analysis results: use the results in step 5 to patch the scenario specification automatically.
7. Inform the user about the amendment of the specification and seek amendment of the requirements from the user.

The C&C analysis process is an iterative and incremental process. After each amendment, the C&C process should be repeated to ensure that the amendment does not introduce new consistency. Tools have

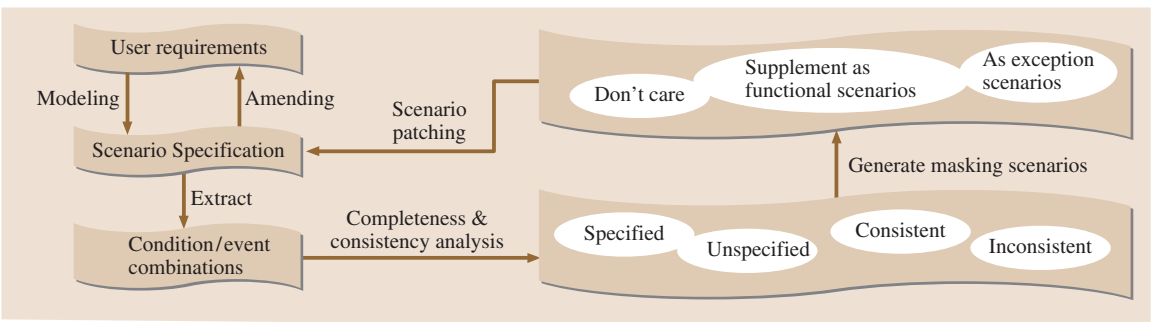

Fig. 24.6 The process of C&C analysis

been developed to perform steps 2–6 automatically. Experiments with the tools in several large, industrial applications have been carried out and the results indicate that the process described above is feasible and scalable to large applications.

24.3.3 Test-Case Generation

Test-case generation techniques can be greatly enhanced by comprehensive formal C&C analysis followed by test-case generation based on Boolean expressions [24.11]. An important distinction of this approach is that test-case generation is based on the quantitative Hamming distance. All previous approaches, including modified condition/decision coverage (MC/DC) and MUMCUT [24.23], were based on user experience and intuition. Exploring the topological hypercube structure of Boolean expressions can easily reveal the faults not discoverable by previous approaches. Furthermore, these two mechanisms can be completely automated, thus saving significant effort and time. After the Boolean-expression generation, the Swiss-cheese test-case-generation tool can be applied to obtain both positive and negative test cases.

The Swiss-cheese (SC) approach is an efficient iterative algorithm developed based on C&C analysis [24.17]. It can identify most error-sensitive positive test cases and most critical negative test cases. Given the Boolean expressions that represent the system specification, the algorithm first maps the Boolean expressions into a multidimensional Karnaugh map called a polyhedron. The algorithm then iteratively identifies all boundary cells of the polyhedron and selects the most error-sensitive test cases among all the boundary cells. The more neighboring negative test cases (degree of vertex – DoV) a boundary cell has, the more error-sensitive it is. The last step is post-checking, which tries to identify critical negative test cases within the polyhedron. For each negative test case, the term Hamming distance (HD) is used to define the minimum different Boolean digits between it and any boundary cells. The HD of all boundary cells is 0. The smaller the HD is, the more critical a negative test case is. It is shown in this paper that negative test cases can detect more failures. The SC approach uses the most critical negative test cases first to test a program, and then randomly chooses the remaining test cases.

24.4 E2E Distributed Simulation Framework

Traditional simulation methodologies adopt a model-code-run approach, such as that used in the Institute of Electrical and Electronics Engineers (IEEE) modeling and simulation (M&S) high-level architecture (HLA) [24.24] and other popular simulation frameworks, which means that the engineers must create a model of the target system, develop the simulation code based on the model, and then run the simulation code, as discussed in GALATEA [24.25]. The E2E scenario-based modeling and simulation framework pro-

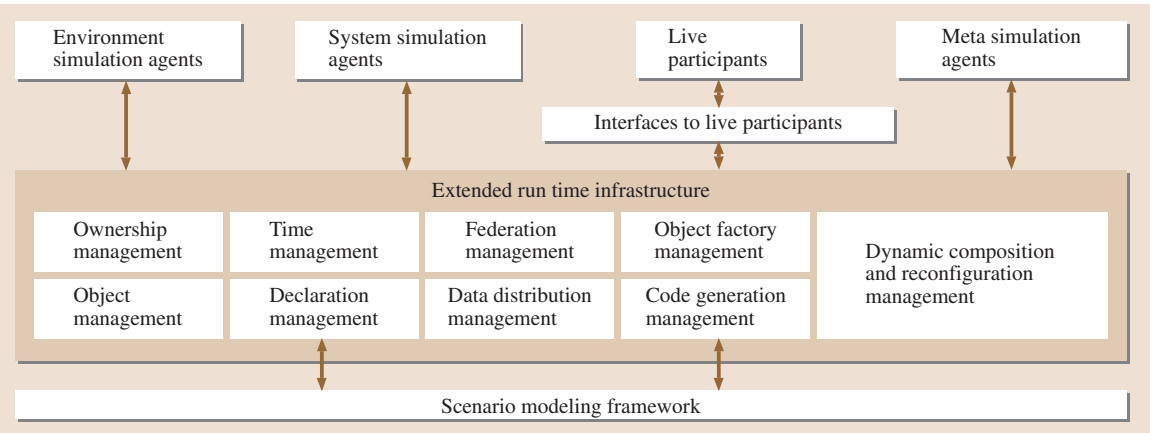

Fig. 24.7 Simulation framework architecture

vides a model-and-run paradigm for simulation. In other words, once the model is available, the model is directly executed for simulation without manual simulation-code development. The simulation code is automatically generated from the system scenario specification. Before the code is generated, the model can be evaluated using existing E2E analysis techniques such as C&C analysis to ensure that the model is correct. Furthermore, the simulation run can be dynamically verified using a formal policy specification.

24.4.1 Simulation Framework Architecture

Figure 24.7 shows the E2E T&E simulation framework running on an SOA, based on software agents. According to the definition in [24.26, 27], a software agent is "an autonomous computer program that operates on behalf of something or someone else" and can "be viewed as a self-contained, concurrently executing thread of control that encapsulates some state and communicates with its environment and possibly other agents via some sort of message passing". The agents here serve as an entity that is capable of carrying out the simulation task and performing a variety of analyses. Both the agents and the simulation framework are designed according to an object-oriented layout to support distribution (of objects/agents), modularity, scalability, and interactivity [24.25], as demanded by the IEEE HLA specification [24.24].

The E2E simulation framework integrates the concepts and tools that support modeling and simulating systems under the distributed, interactive, continuous, discrete, and synthetic focuses. The simulation framework consists of:

- The ACDATE scenario language and the framework that allows the construction of the system models.
- The on-demand automated dynamic code generator that supports rapid and automated simulation/real system-code generation such that simulation can be carried out once the system model is ready. No additional programming effort is needed. The *execution* here means that the real system components' execution is involved in the system simulation, i. e. end-to-end simulation including the end hardware-in-the-loop and man-in-the-loop.
- Simulation agents that carry out the simulation tasks and form a simulation federation (in the HLA sense) serve as the simulator for the whole system. These agents can be geographically distributed on computers that are interconnected via a local-area and/or wide-area network [24.19].
- An extended runtime infrastructure to support the agents' work. As required in [24.28], the simulation here is separated from the target system model, which makes the simulation framework flexible and generic.

As discussed in [24.24], traditional simulation techniques should be extended to support interactive simulation of a number of programs executing in heterogeneous and distributed computers that interact with each other through communication networks and are managed by a distributed operating system. The IEEE has provided the HLA framework to allow the development of a standard simulation framework with many different simulation components, which is used as a reference for the design of our framework. Figure 24.7 shows the architecture of our simulation framework. As can be seen, the scenario modeling framework provides the scenario specifications of the target systems. The extended runtime infrastructure separates the simulator (which consists of the agents and/or the live participants) from the target system model and provides the necessary runtime support for the simulator.

24.4.2 Simulation Agents' Architecture

The E2E T&E simulation framework is object-oriented, agent-based, discrete-event-driven, distributed, and real-time. In object-oriented terms, E2E simulation is based on the integrated ACDATE scenario model, which is based on SoS/SOA and the object-oriented modeling methodology. Each component in the system is modeled as a specific object–actor that has interfaces (actions), behaviors (scenarios) and constraints (policies). The simulation is carried out by a set of simulation agents. The agents are the most important elements in our simulation framework.

An agent can simulate either a single actor or multiple actors. Two agents may or may not reside in the same computation site. Agents can talk with each other via standard communication protocols. The behavior of a simulation agent is determined by the SoS/SOA scenario model of the actors simulated on this simulation agent. Based on the system's scenario model, it is clear how an actor will behave to some outside stimulus either from the environment or from some other agents under given conditions. The outside stimuli are modeled as discrete events that can be received and processed by an actor. Once an event arrives at an actor, the ac-

tor will put it into its waiting queue. How the events in the waiting queue are processed depends on the actor's scenario model, i. e. if the system is modeled as a multi-tasking actor, any incoming event can be processed as long as there is enough resource. If the system is modeled as a single-tasking actor, an incoming event can be processed only if no other task is scheduled to use the processor, and so on. Due to some uses of the framework for decision-making, one simulation run should finish before a given deadline, if required.

The simulation can be formally described as follows. Simulation of an actor A_i starts from the point when an event $E_{i,k}$ arrives at A_i. At the point, A_i will pick up scenario $\text{Scnr}_{i,w}$, which describes the behavior of A_i in response to $E_{i,k}$ and sends $\text{Scnr}_{i,w}$ to a scenario simulation/execution engine, as shown in Fig. 24.8. The scenario simulation engine will then interpret $\text{Scnr}_{i,w}$ and perform the following:

- Check current system condition, which includes the A_i own conditions $\{C_{i,i1}, C_{i,i2}, \ldots, C_{i,im}\} \in \{C_{i,0}, C_{i,1}, \ldots, C_{i,Mi}\}$ and/or other actors' conditions as guard conditions.
- Based on the system condition and chosen scenario $\text{Scnr}_{i,w}$, the simulation engine will choose an execution path which includes a series of actions $\{\text{Act}_{i,v1}, \text{Act}_{i,v2}, \ldots, \text{Act}_{i,vn}\}$.
- The scenario simulation engine will carry out the chosen actions, whose semantics are also specified using scenarios. Thus, whether an action can be successfully performed also depends on the system conditions at that point. An action may change the owner actor's status by changing the values of the data owned by the actor; or emit a new event either to other actors or to its owner actor.
- Agents' states will be changed accordingly as the actions are performed, which is reflected in the data-changing function: Act: $D \rightarrow D_0$, where D is the set of data values before the action Act is performed, where D_0 is the set of data values after the action Act is performed.

Events are the only channel through which different actors can communicate with each other. An event can carry parameters to provide more information for the receiver to make decision on how to respond to the incoming event. Simulation agents used here contain versatile communication capability, which is implemented by the communication component of each agent, and thus an agent can be exposed to the outside world as a traditional transmission control protocol/internet protocol (TCP/IP) service, a dedicated network component,

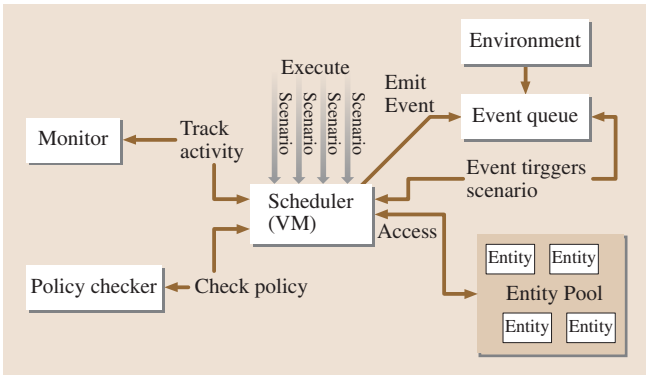

Fig. 24.8 Simulation engine inside a distributed simulation agent

or more generally a web service. In the latter case, as each simulation agent is exposed as a web service, it is easy for the simulation users to perform the simulation tasks on the internet.

There are three major types of simulation agents: environment simulation agents, system simulation agents and meta-agents. By separating the environment simulation agents and system simulation agents, it is easy to study the target system's behaviors under different environments without touching the target system model and simulation, which only requires a change to different environment simulation agents in the simulation. Meta-agents are agents that monitor and coordinate the whole simulation. With the help of these meta-agents, engineers can easily know what is going on in the distributed simulation from a global point of view. These meta-agents can also help perform dynamic analyses that involve more than one participating simulation agents such as the generation of overall system behavior.

24.4.3 Simulation Framework and Its Runtime Infrastructure (RTI) Services

The simulation extended runtime infrastructure is an extension and enhancement of high level architecture/runtime infrastructure (HLA/RTI) [24.24], which serves as a design reference for our simulation framework's runtime infrastructure. The major improvements are the automated ACDATE/scenario code generation and deployment management, event management on SOA, and automated simulation runtime reconfiguration and recomposition. With the help of these services, our simulation framework is capable of providing on-demand simulation, which means that the simulation code can be dynamically obtained and used for simulation from the dynamic code generator whenever it is

demanded by the users; as well as dynamic simulation reconfiguration.

In contrast to traditional HLA/RTI services, which are exposed as traditional remote procedure call (RPC) methods using the user datagram protocol (UDP)/TCP, the services provided by our simulation framework can be exposed as either RPC-like service using binary communication data via TCP, or WS using simple object access protocol (SOAP) messages to carry communication data via the hypertext transfer protocol (HTTP).

Managing events in SOA

The simulation framework is developed on top of an SOA, and thus it can reuse resources on SOA, a lot of benefits can be obtained. One of these is that a simulation agent does not need to know the existence of other simulation agents. Simulation framework RTI will provide an event-space service (ESS) to facilitate communication among simulation agents, as shown in Fig. 24.9. The services provided by ESS include:

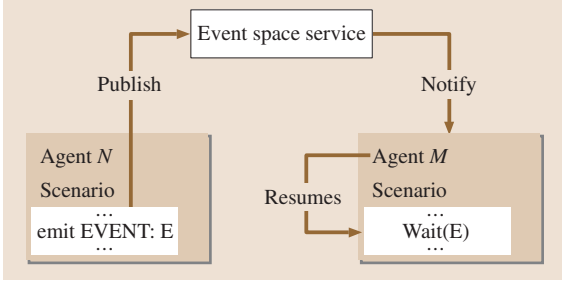

Fig. 24.9 Event publishing and notification example

- Event registration
  - Event publishing registration: before sending out any event, agents must register what events they will send out with the ESS.
  - Event subscription registration: an agent must subscribe the interested events before it can actually know that the event happens.
- Event publishing: agents can emit events using ESS
- Event notification: ESS can notify the occurrence of events to those agents that have subscribed the events.

Automated Simulation Code Generation and Deployment

The simulation code is generated based on the scenario specification, which includes the ACDATE definition and scenario description, as shown in Fig. 24.10. Each ACDATE element will be translated into an object with the attributes defined in the specification. Instrumentation code will be inserted into the objects to interface with the monitor and policy checker. Each scenario will be translated into a procedure that is basically a sequence of operations on the ACDATE objects or emitting events. Similarly, instrumentation code will be inserted into the procedure so that the procedure can interface with the scheduler to schedule concurrent execution and the event queue for emitting new events.

Table 24.2 shows a sample simulation code automatically generated with instrumentation code that interfaces with the scheduler, event queue, monitor, and policy checker.

Simulation framework provides two base components for scenario code generation: BaseACDATE

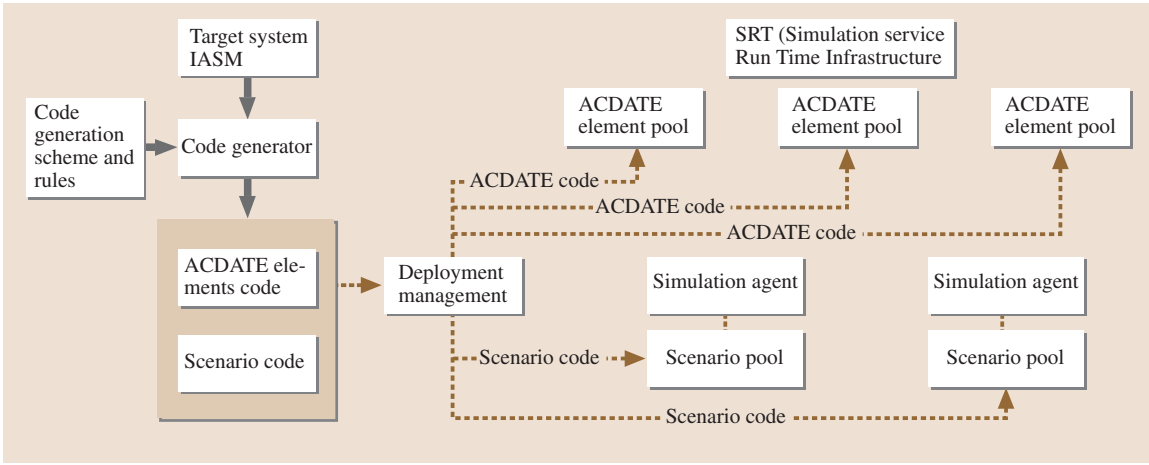

Fig. 24.10 Automated simulation-code generation and deployment

Table 24.2 Automatically generated code example

```
public override void ScnrFunc()// a scenario
 . . . . . .
{
Condition_4 condition4 = new Condition_4();// obtain ACDATE elements
Action_5 action5 = new Action_5();
Data_2 data2 = new Data_2();
e = newSimRunTimeLogArgs(SimRunTimeLogArgs.LogTypes.ScnrPreStatement,2,
1, 0, "Before Step 1 in scenario 6");
this.OnPreScnrStatementEventHandler(this, e);
data2.SetValue(1); // data2's value is now changed to integer 1
e = newSimRunTimeLogArgs(SimRunTimeLogArgs.LogTypes.DataWrite,
2, 1, 0, "After Step 1 in scenario 6");
// interface to instrumentations such as policy checker embedded here
this.OnPostScnrStatementEventHandler(this, e);
System.Windows.Forms.MessageBox.Show("Agent 1 - Step 1 done");
e = newSimRunTimeLogArgs(SimRunTimeLogArgs.LogTypes.SchedulingFlag,
System.Threading.Thread.CurrentThread.GetHashCode(), 1, 0, "Calling
Scheduling");
this.OnSchedulingEventHandler(this, e);
System.Threading.Thread.CurrentThread.Suspend();// interface to simula-
tion scheduler
 . . . . . .
}
```

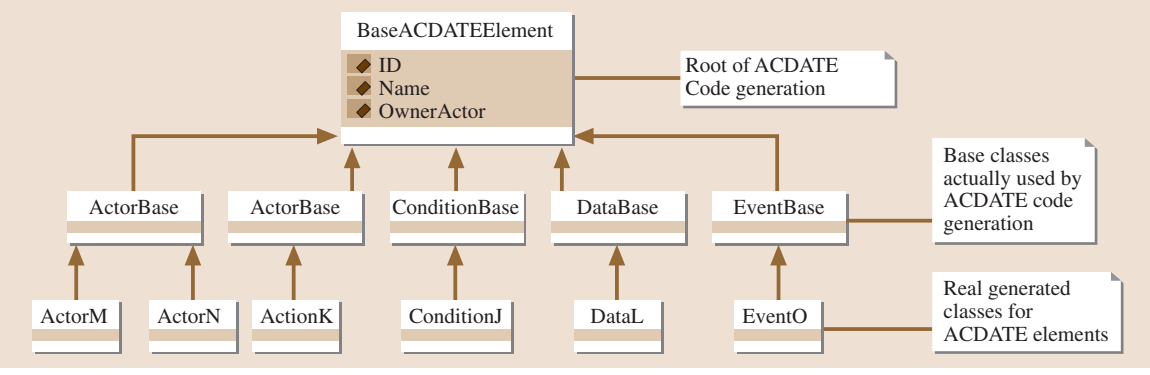

Fig. 24.11 Generated ACDATE code hierarchy

(Fig. 24.11) and BaseScenario (Fig. 24.12). The BaseACDATE component contains all the base class definitions for the ACDATE elements in the scenario. The BaseScenario component provides the base class for the scenario specification in the scenario model while referencing the used ACDATE elements' information in the BaseACDATE component.

For different system simulations, different simulation code will be automatically generated based on the target system's scenario model. The generated code is divided into two major categories of components: ConcreteACDATE and ConcreteScenarios. ConcreteACDATE here does not mean a single component but a collection of components holding

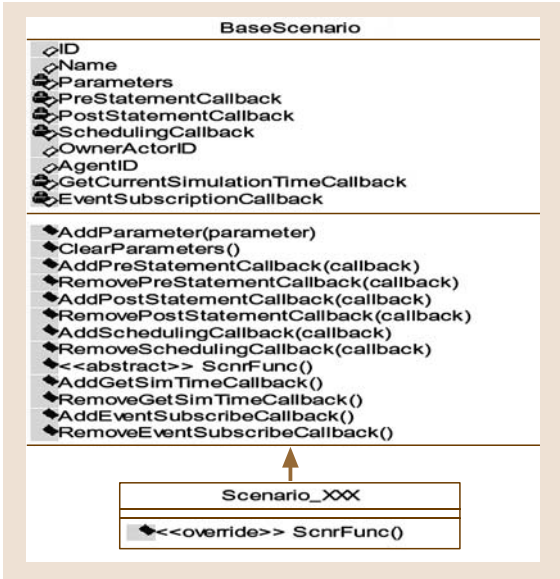

Fig. 24.12 Generated scenario code hierarchy

the generated code for the concrete ACDATE elements in the target system's scenario model. Similarly, ConcreteScenarios is a collection of components holding the concrete scenario code of the target system's scenario model, with only references to the related ConcreteACDATE components. Each generated component has its own deployment configuration document, based on which the deployment management will deploy the simulation code properly. More details of automated simulation code generation will be discussed in later sections.

At simulation runtime, with different simulation code loaded, the simulation agents can simulate different target systems while keeping the simulation agents themselves untouched.

In a distributed simulation system, simulation deployment is an important issue. Traditionally simulation deployment is done manually, which is time-consuming and error-prone, especially for large simulation systems. Simulation deployment can be formally specified and then be automated. As has been discussed, with different target system models (code) loaded, the simulation agents can perform different simulation tasks. Machine understandable simulation deployment specification [in extensible markup language (XML)] will be provided. Service RTI will provide a dynamic model (code) load/unload service based on the simulation deployment specifications.

Automated Simulation Reconfiguration and Recomposition

For a large-scale distributed simulation, a single failure may corrupt the whole simulation if the failure point is critical (a single point of failure). It is important to organize the simulation services (agents) distributing across the internet to form a functional simulator, which can make use of the unutilized computation power. With the policy and dynamic reconfiguration service (DRS), the simulation framework should be able to see the change of system behavior when a new policy becomes effective during simulation without shutting down the system.

Dynamic simulation composition and reconfiguration management involves the following issues:

- Automated simulation-agent deployment and discovery
- Automated simulation-agent status monitoring and failure detection
- Automated dynamic simulation code generation
- Automated dynamic simulation configuration generation
- Automated dynamic simulation deployment and redeployment.

A simulation agent knows nothing about the target system until the corresponding simulation code is loaded into the agent. With different simulation code loaded, the simulation is capable of simulating different target systems. The simulation agent can also unload the previously loaded simulation code component and reload a new set of simulation code components to simulate another target system. In this sense, the real components of a functional simulation are the dynamically and automatically generated simulation distributed across the network.

The first problem one may face is how the dynamically generated simulation code components can know where the counterparts and the runtime infrastructure services are. This can be solved with using a dynamically and automatically generated configuration of the simulation. With any given simulation topology, users can specify where and how a simulation should be deployed. A configuration document will then be automatically generated for each dynamically and automatically generated simulation code component based on the users' deployment requirements, such that each simulation code component can know where to obtain the required resources and services, as well as how to communicate with the runtime infrastructure services and its counterparts. This has been introduced in previous sections.

The automatically generated simulation code components along with their configuration documents are deployed based on the deployment configuration documents to set up the simulation agents properly with the help of the automated deployment management services, as shown in Fig. 24.13.

During the simulation, the status-monitoring services continuously monitor the status of the simulation agents involved. Once the failure of a simulation agent is detected, the runtime infrastructure will try to discover an available simulation agent and perform the code and configuration generation again for the alternative agent. The new simulation agent is then loaded with the simulation code and configuration and the simulation is resumed. Using automated simulation composition and deployment, users can also easily change the deployment of the simulation anytime. However, in these cases all the unfinished work on the crashed simulation agents will all be lost, if it has not been saved.

There is another scenario in which dynamic simulation reconfiguration can be used: on-the-fly model changing and continuous simulation. Users can change the target system model during the simulation without restarting the simulation to reflect the effect of model modification.

Once the target system model has been changed, based on the original users' deployment requirements and the status of the simulation agents, the automated simulation deployment service can determine which agents are affected by the model modification. The simulation code components and configuration documents are then regenerated based on the modified model. Once the affected simulation agents enter a safe state where unloading and reloading simulation code components will not affect other running simulation agents, the automated simulation deployment service will unload and reload the simulation agents with the modified simulation code components. Before the simulation components are unloaded, the status of the simulation agents is saved. The saved information is used to restore the original agents after the new simulation code components are reloaded. In this way, the users' simulation will not be interrupted by the model modification.

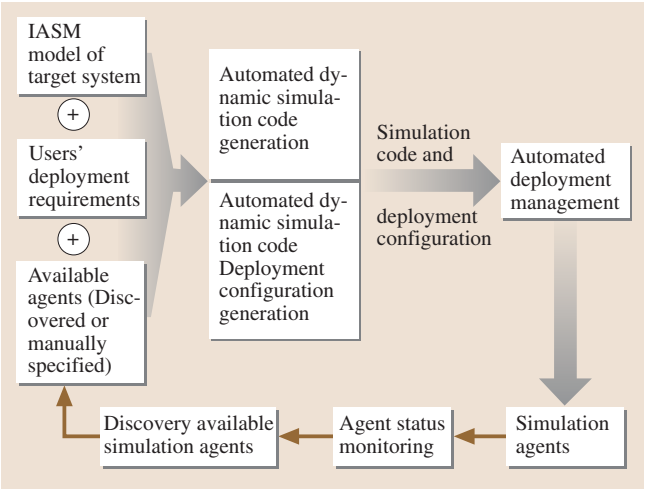

Fig. 24.13 Automated simulation reconfiguration and recomposition

24.5 Policy-Based System Development

Policies have been increasingly used in computing systems for specifying constraints on system status and system behaviors. A *policy* is a statement of the intent of the policy maker or the administrator of a computing system, specifying how he or she wants the system to be used [24.29]. Usually, policies are hard-coded into the system implementation, For instance, if a policy states that "passwords must be at least eight characters long", there must exist a snippet of code in the system implementation that checks the length of passwords. Hard-coded policies can cause major problems for the system such as:

- It is difficult and expensive to update.
  Whenever a policy needs to be changed (e.g. the system administrator wants to reduce the minimum length of valid passwords from eight characters long to five characters long), the whole system has to be shut down, and the code has to be modified, recompiled, and redeployed. The process is lengthy and significantly increases an organization's operating expenses. Shutting down a mission-critical system, in most cases, is prohibitive and may cause disastrous consequences to the mission.
- It is difficult to manage.
  Hard-coding policies do not separate policy specification from system implementation. Policies are spread throughout the system implementation. If a policy maker wants to know how many policies

there are in the system, or what are the policies that are defined for the role of supporting arms coordinator, there is no easy way to find the answers.

In the past decade, a number of *policy specification languages (PSLs)* have been proposed [24.29–33]. PSLs provide a simple and easy-to-use syntax to specify policies separately from the system implementation. A user interface can provide a means for policy makers to specify and manage policies. A policy engine can interpret and enforce policies at runtime rather than at compilation time, which allows policies to be dynamically added, removed, updated, and enabled or disabled.

24.5.1 Overview of E2E Policy Specification and Enforcement

Figure 24.14 shows the simulation environment that we developed for highly developed systems. When developing such systems, V&V needs to be performed at each step of the development. First, the system requirement is translated into the formal specification. The specification is verified by a C&C check. After several iterations of verification, the final specification is obtained. Test cases are generated from the specification. An automated code-generation tool generates the executable for simulation. This process has been reported in [24.17].

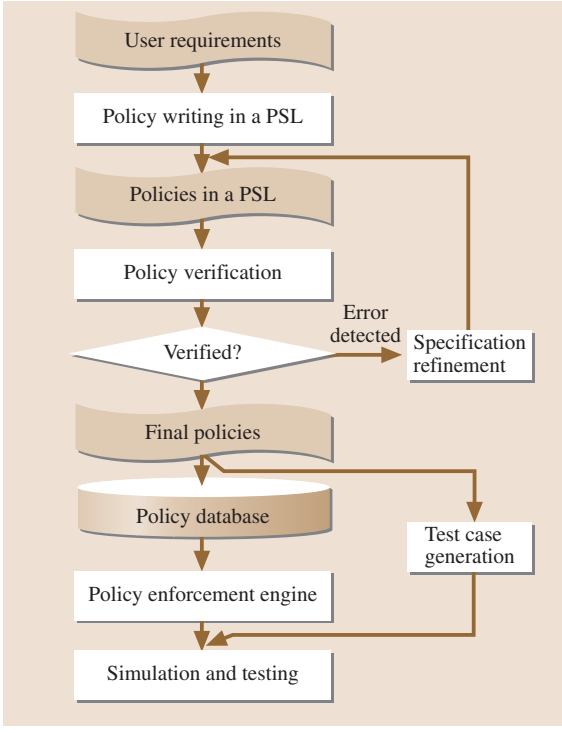

Fig. 24.14 The policy specification and enforcement architecture

Independently of the development processes, the policies are extracted from the requirements and then written in a policy specification language (PSL). Similarly to the specification, the policies are verified by a C&C check to detect any incomplete and inconsistent policies. After the policies pass the verification, they will be stored in a policy database. Test cases that dynamically check C&C can be generated from the final policies. During the course of simulation execution, a policy-enforcement engine dynamically loads the policies from the policy database, interprets them, and enforces them at runtime. Since the policy engine dynamically interprets and enforces policies, policies can be easily changed (added, removed, updated, and enabled/disabled) on-the-fly at any time.

This paper not only makes use of the flexibility of policies, but also applies policies as a dynamic V&V method. In fact, many functional requirements of a system can be extracted as policy requirements. For example, array index range check, probability value range check, and execution orders can be written as policies. As a result, the policy enforcement can dynamically check the validity of computing. Policy enforcement is particularly useful in detecting those bugs that are difficult to catch during unit testing and those with complicated interactions due to concurrent threads and processes in simulation. Note that traditional testing suffers from the need to set up the environment to a given state and run the program to see if the program behaves as intended, which is time-consuming and difficult. Policy is a good way of ensuring the simulation program is correct because an engineer can specify any kinds of policies that need to be enforced to see if the simulation program performs correctly. Another advantage of using simulation to run policies is that simulation can run extensive cases to ensure extensive coverage. Thus, we can have both static and dynamic coverage, e.g., how many times a specific set of scenarios have run, and how many times a specific scenarios will happen, and how many of these scenarios are performed correctly.

24.5.2 Policy Specification

We designed the policy specification and enforcement language (PSEL) that covers obligation policies, authorization policies and system constraints. A policy editor

and a graphical policy-management interface have been developed for policy input. Obligation policies and authorization policies are defined on roles rather than on individual actors. A role represents a management position and the responsibilities and rights associated with that management position. Actors are assigned to roles according to their management positions. Since actors take particular roles in an organization, policies specified on a role will in turn apply to actors who take this role.

Obligation policies define a role's responsibilities, specifying what actions a role must or must not take under a condition. *Positive obligation policies* are event–condition–action (ECA) rules with the semantics that: on receiving a triggering event E, a role R must perform the action A if condition C is true. For instance, the policy "on receiving the call for fire, the supporting arms coordinator must issue a fire order" should be defined as a positive obligation policy, since it specifies the responsibility of actors who take the supporting arms coordinator role.

Negative obligation policies forbid a role from performing action A if condition C is true. For instance, the policy "If a main battle tank can shoot, it must not reject the fire order" should be defined as a negative obligation policy. If violated, the policy enforcer will inform the system simulator of the detection of the policy violation. The system simulator will perform the compensation action that is intended to minimize the consequences caused by the policy violation. Table 24.3 gives the syntax and examples of obligation policies.

Authorization policies define a role's rights to perform actions, specifying which actions are allowed or prohibited for the role under a certain circumstance. In PSEL, authorization policies are specified and enforced through access control models. Currently, two access control models are supported in PSEL: the Bell–LaPadula (BLP) model and the role-based access control model.

Bell–LaPadula (BLP) model [24.34] is a mandatory access control model widely used in military and government systems. It controls information flow and prevents information from being released to unauthorized persons. The BLP model defines four ordered *security levels*: Unclassified < Confidential < Secret < Top Secret. Security levels are then assigned to actors and data. An actor's security level is called the *security clearance*; data's security levels are called the *security classification*. Each action in the system has a subject (an actor) that performs the action, data (objects) on which the action is performed, and an accessing attribute that indicates the nature of this action (read, write, both, or neither).

The BLP model defines two access rules. The *no-read-up* rule applies to all actions whose accessing attributes are read. It specifies that an actor is not allowed to read data if the actor's security clearance is lower than the data's security classification. For instance, the observer in the special operations forces (SOF) team, with a security clearance of confidential, is not allowed to read (attribute: read) the target destroyed report (security classification: secret). The no-read-up rule prevents unauthorized persons from reading information they are not supposed to read. The *no-write-down* rule applies to all actions whose accessing attributes are write. It specifies that an actor is not allowed to write data if the actor's security clearance is higher than the data's security classification. The no-write-down rule prevents actors with higher security clearance from accidentally

Table 24.3 Examples of obligation policies

Policy type	Syntax	Example

Positive obligation policy	MUSTDO {
	definedOn ROLE
	triggeredBy EVENT
	do ACTION
	on CONDITION
	}
		MUSTDO {
		definedOn ROLE:SupportingArmsCoordinator
		triggeredByEVENT:ReceiveCFF
		do ACTION:IssueFireOrder
		on CONDITION:
		}

Negative obligation policy	MUSTNOTDO {
	definedOn ROLE
	do ACTION
	on CONDITION
	perform COMPENSATION
	}
		MUSTNOTDO {
		definedOn ROLE:MainBattleTank
		do ACTION:MBTRejectMission
		on CONDITION:MainBattleTankCanShoot
		perform COMPENSATION:Warning
		}

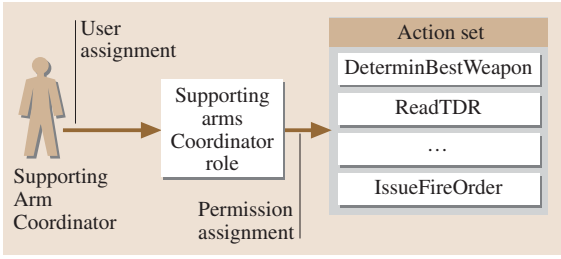

Fig. 24.15 The RBAC model

writing classified information to an unclassified media, so that unauthorized persons can read the information.

The *role-based access control (RBAC)* model [24.35–37] has been increasingly implemented in various systems, due to its policy-neutral nature. As shown in Fig. 24.15, in RBAC a group of roles are defined according to the semantics of a system. Actors are then assigned to roles according to their management position in this system. A set of actions are then assigned to a role, giving it the permissions to perform these actions. PSEL also supports a role hierarchy. *A role hierarchy* represents the superior and subordinate relationships among roles, allowing a superior role to obtain all permissions of its subordinate roles automatically. *Role delegation* is also supported, which enables a role to temporarily transfer its permissions to other roles, and for them to be revoked at a later time.

The access rule defined in the RBAC model is simple: an action A is allowed if

- there exists a role R, such that A.owner takes R, and
- R has permission to perform A.

For instance, the supporting arms coordinator (who takes the supporting arms coordinator role) is allowed to issue the fire order. Figure 24.16 shows the graphical user interface for role-based authorization policy specification.

System constraints define constraints on system status and behaviors that must hold in the system execution or simulation. *System constraints on data* specify that data must or must not be within a certain range. For example, the policy that "the distance between the SOF team and the surface-to-surface missile (SSM) launcher must be $\geq$ 3000 feet at all times" is a system constraint on data. *System constraints on actions* are currently temporal logic on actions. Examples could be "action DetermineBestWeapon must occur before action InputFireOrder" or "the call for fire (CFF) command can

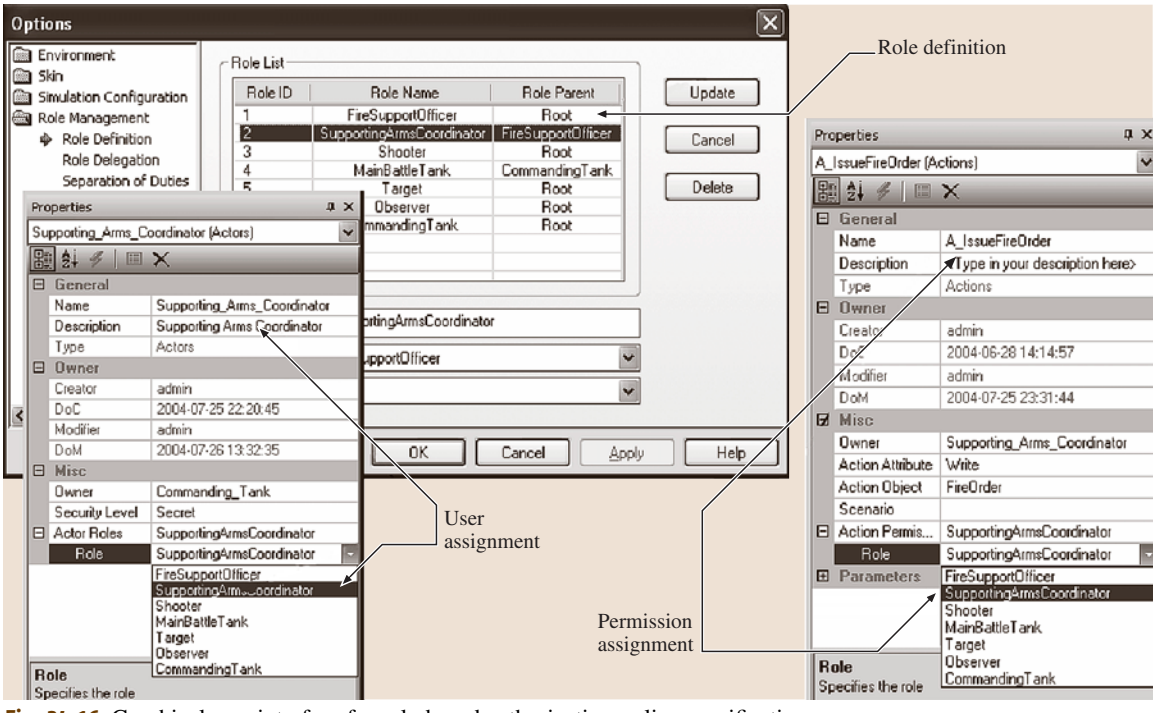

Fig. 24.16 Graphical user interface for role-based authorization-policy specification

Table 24.4 Examples of specifying system constraints

Policy type	Syntax	An example

System constraints on data	MUSTBE / MUSTNOTBE {
	appliedTo DATA
	status EXPRESSION
	on CONDITION
	perform COMPENSATION
	}
		MUSTBE {
		appledTo DATA:SOFTeamDistanceFromSSM
		status EXPRESSION:">=3000"
		on CONDITION:TRUE
		perform COMENSATION:SOFRetreat
		}

System constraints on actions	MUSTBE / MUSTNOTBE {
	do ACTION:Operator
	(action parameters)
	on CONDITION:TRUE
	perform COMPENSATION
	}
		MUSTBE {
		do ACTION:Sequence
		(ACTION:DetermineBestWeapon
		ACTION:InputFireOrder)
		on CONDITION:TRUE
		perform COMPENSATION:Warning }

be issued only once". Currently, the following temporal logic operators are supported by our ACDATE-based policy framework:

- Concurrency (A, B): A, B occur concurrently;
- Sequence (A, B, C): A, B, C occur in this sequence;
- Order (A, B, C): A, B, C occur in this order consecutively;
- Either (A, B): either A occurs or B occurs, but not both;
- Exist (A): A must occur;
- Once (A): A must occur once, and only once.

Conditions are associated with system constraints, specifying when these policies are to be enforced. In addition, compensation actions are defined in a system constraint. When policy violation is detected, associated compensation actions will be performed by the simulator to minimize the consequences brought about by policy violation. Table 24.4 gives syntax examples of specifying system constraints.

24.5.3 Policy Enforcement

Policies are enforced in the course of system simulation. In the initialization phase of system simulation, the policy enforcer will load policies out of the policy database and register them with the system simulator according to their semantics. Policies are registered so that the system simulator knows when to trigger policy enforcement. The system simulator triggers the policy enforcer when a registered event occurs, a registered action is performed, or a registered datum is modified. The policy enforcer will enforce relevant policies attached to these registered events, actions, or data, and return the results of enforcement back to the system simulator.

Policy enforcement can be classified into three categories: policy checking, policy execution, and policy compensation. *Policy checking* verifies if policy violations are detected when actions are performed or data are changed. *Policy execution* executes the action defined in the policy when receiving the triggering events. *Policy compensation* executes the compensation action defined in the policy when policy checking detects a policy violation. All checkable policies come with a compensation action.

Only positive obligation policies are executable policies. The other types of policies (e.g. negative obligation policies, all authorization policies and all system constraints) are all checkable policies. *Executable policies* influence the paths of system simulation through the actions defined in them. When the triggering event occurs, executable policies registered to this event will be enforced, and the action specified in the policy specification will be executed by the system simulator. *Checkable*

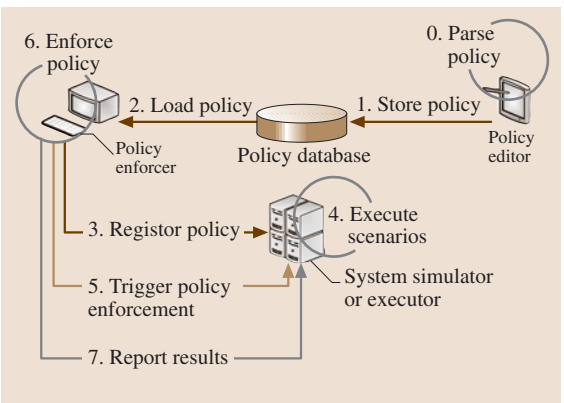

Fig. 24.17 The policy enforcement framework

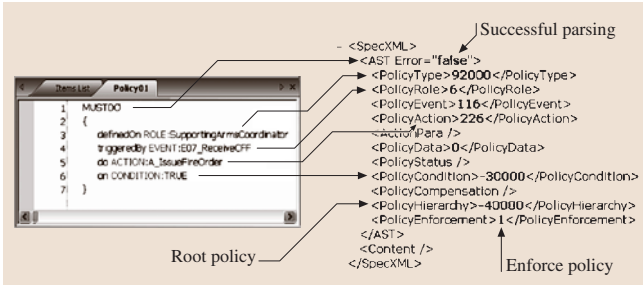

Fig. 24.18 Policy parsing

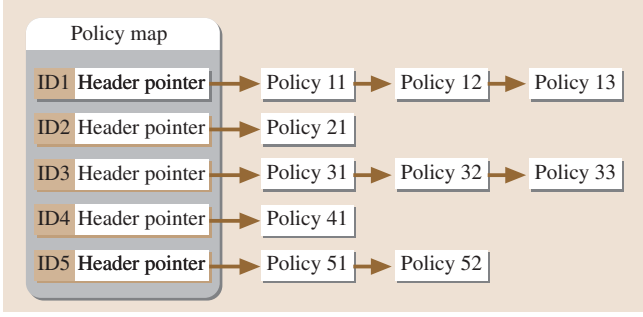

Fig. 24.19 Policy map

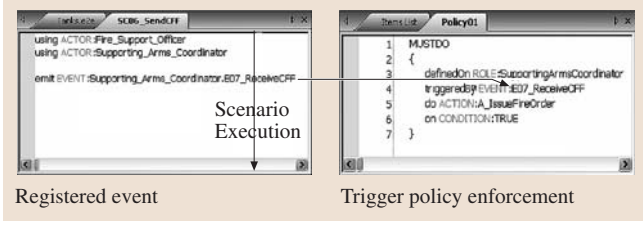

Fig. 24.20 Policy enforcement triggering – event

policies can also influence the paths of the system simulation through compensation actions. When data are changed or actions are performed, checkable policies registered to the data or actions will be enforced. If policy violations are detected, the compensation action specified in the policy specification will be executed by the system simulator.

Figure 24.17 illustrates the policy enforcement framework. After ACDATE elements are defined and policies are extracted, policies are specified in the policy editor, which parses policies for correctness, C&C, and stores them in the policy database. During the initialization phase of simulation, the policy enforcer loads policies from the policy database, interprets them and registers them with the system simulator. While the simulator executes system scenarios, it triggers the policy enforcement when registered events occur, registered actions are performed, or registered data are changed. The policy enforcer checks or executes policies and returns the results to the simulator. Based on the returned results, the simulator determines what the next system scenarios are.

The policy editor parses policies for correctness and C&C. On successful parsing, the policy editor translates policies into XML and stores them in the policy database. The policy parser is implemented by: another tool for language recognition (ANTLR). According to the policy syntax, ANTLR creates an abstract syntax tree (AST) for each policy. Policy elements (roles, actions, condition, etc.) are extracted by traversing the tree, and translated to the XML representation, as shown below. The XML representation of a policy is then stored in the policy database as a string. Figure 24.18 shows an example policy parsing.

The purpose of policy registration is to let the system simulator know when policy enforcement should be triggered. In our ACDATE-based policy framework, three out of the six ACDATE elements can trigger policy enforcement: event, action, and data. Events occurrences will trigger the enforcement of positive obligation policies; action performances will trigger the enforcement of negative obligation polices, authorization policies, and system constraints on actions; data changes will trigger the enforcement of system constraints on data, as shown in Table 24.5.

To improve performance, a policy map is created, mapping a particular event, action or data to a list of relevant policies to which it is registered, as shown in Fig. 24.19. All policies registered to a particular action,

Table 24.5 Policy registration

Policy type	Authorization policy	Positive obligation policy	Negative obligation	System constraints on data	System constraints on action
Event		X			
Action	X		X		X
Data				X	

event or datum form a linked list. The linked list and the identification (ID) of the action, event or datum are then organized as a policy map. When policy enforcement is triggered, the policy enforcer locates the policy linked list of a particular action, event or datum, and enforces all policies in the list.

After policies have been registered, the simulator initializes the data and starts running the system scenarios. The simulator keeps an eye on the simulation of system scenarios, and triggers the policy enforcement by invoking the EnforcePolicy method in the policy enforcer whenever an event is triggered, an action is performed or a datum is changed. Figure 24.20 shows an example where a triggering event causes a policy enforcement execution. The policy enforcer enforces all relevant policies, records all violations in the policy log, and returns the policy log back to the system simulator.

When policy enforcement is triggered, the simulator invokes the EnforcePolicy method in the policy enforcer, passing the ID of the event, action or datum that triggered the policy enforcement. On being triggered, the policy enforcer looks into its policy map, maps the ID to a list of policies to which it has been registered, and enforces them one by one. Authorization policies are not registered in the policy map, and they are enforced before obligation policies and system constraints are enforced. When policies are being enforced, all violations are recorded into a policy log that is returned to the simulator. The EnforcePolicy method returns a Boolean value indicating whether policy violations are detected. Figure 24.21 gives the policy enforcement algorithm.

```
bool CPolicyEnforcer::PolicyEnforcer(const long lEntityID)
{
    //  lEntityID is the ID of the entity that triggers policy enforcement
    //  An Entity could be an avent, action, or data
    if (GetEntityType(lEntityID) == ACTION_TYPE)
    {
        Verify whether this action is allowed by BLP  Model;
        Verify whether this action is allowed by RBAC Model;
    }

    deque<CPolicy*> pDequePolicy = m_mapPolicyMap.find(lEntityID);
    for (size_t i = 0; i < pDequePolicy.size(); i++)
    {
        bool bPassed = true;
        CPolicy* pPolicy = pDequePolicy[i];

        switch(pPolicy->m_lPolicyType)
        {
        case POLICY_MUSTDO:
            bPassed = PolicyCheckMustDo(pPolicy, lEntityID);
            break;
        case POLICY_MUSTNOTDO:
            bPassed = PolicyCheckMustNotDo(pPolicy, lEntityID);
            break;
        case POLICY_MUSTBE:
            bPassed = PolicyCheckMustBe(pPolicy, lEntityID);
            break;
        case POLICY_MUSTNOTBE:
            bPassed = PolicyCheckMustNotBe(pPolicy, lEntityID);
            break;
        }

        if (!bPassed)
        {
            Record Policy Violations;
        }
    }

    if (no policy violations) return true;
    return false;
}
```

Enforce authorization policies

Enforce obligation policies

Enforce system constraints

Fig. 24.21 Algorithm for policy enforcement

24.6 Dynamic Reliability Evaluation

Software reliability has been defined as the probability that no failure occurs in a specified environment during a specified (continued) exposure period. Existing software reliability models assess reliability statically in the development process. The E2E T&E perform dynamic evaluation at runtime using a software reliability model that is integrated into the ACDATE scenario model. Figure 24.22 illustrates the development and operation processes using this model.

From the scenario specification, the atomic components can be identified and a data collector is instrumented around each atomic component, which collects runtime failure data during testing and operation. Based on the collected data, the reliability of both components and the SoS can be assessed. The rest of the section explains the major components in this reliability assurance process.

24.6.1 Data Collection and Fault Model

An SoS may consist of many subsystems or components and it is hard to exactly distinguish their contributions to the overall reliability of the SoS due to the anfractuous dependency relations among them. A component may consist of several subcomponents, in which case its reliability can be computed analytically, provided the reliability of each subcomponent is known. The breakdown can be continued to each subcomponent. However it must end somewhere when the component is either indivisible or it is not worthwhile dividing it further. We then consider these components as black boxes or atomic components in our reliability model. In other words, the reliability of an atomic component is not the result of (but an input to) our reliability model.

Although the decision on what component shall be treated as a black box is truly application-dependent,

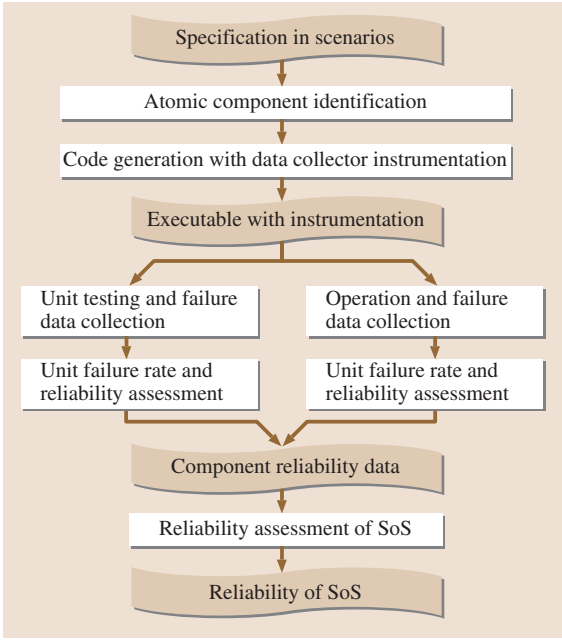

Fig. 24.22 Dynamic reliability assurance

we propose three general principles to curb arbitrariness and strengthen the rationale for our reliability model.

Granularity principle: a finer granularity can lead to more accurate evaluation results but may increase the complexity of the computation. Basically there is a trade-off between granularity and accuracy and their balance depends on the specific application and its requirements.

Perceptible principle: if a component is treated as a *white box*, the opposite of a black box, then the dependencies among all its subcomponents that have an effect on reliability will be modeled explicitly and hence be accounted for during the reliability computation. If the dependency is neither clear nor completely modeled, then the result of the computation will be biased.

Continuous principle: a component is a white box only if its super-component is a white box. It is of no benefit if a black box has a subcomponent as a white box since the reliability of a black box is not computed.

The perceptible principle and the continuous principle define the *effective domain* of the ACDATE scenario model. Inside the domain, everything is explicitly modeled and hence is a white box, while a component outside the domain is a black box. Code that is either manually developed or automated generated based on the model represents its effective domain in the system.

The black box, which is outside the effective domain of the model, can be further categorized as follows:

- Operation on hardware through a device driver;
- A system call provided by the operating system;
- A method or attribute in the programming platform, e.g., the vector class in Java, C# or C++ standard template library (STL);
- A method or attribute in a library provided by a third party;
- Input from a human operator;
- A component in a remote location.

Different types of black boxes incur different reliability estimations, which will be detailed in the next subsection.

Figure 24.23 illustrates the effective domain, where a circle is a white box, a disk is a black box, a line is a dependency relation, and a dashed line is an unclear dependency relation.

To collect failure data, each function call to an atomic component is replaced by a wrapper call that collects failure data related to the atomic component.

Assume a call to an atomic component is atomfun(p1, p2, . . . , pn), where atomfun is the function name, and p1, p2, . . . , pn are the parameters. In the data collector instrumentation stage in Fig. 24.22, the function call is replaced by a wrapper function call: dataanalyzer(atomfun, p1, p2, . . . , pn).

The data collector function is:

```
dataanalyzer(atomfun, p1, p2, . . . , pn)
{
Increment the execution counter;
Find the specification of atomfun;
Verify the legitimacy of p1, p2, . . . , pn;
Call atomfun(p1, p2, . . . , pn);
```

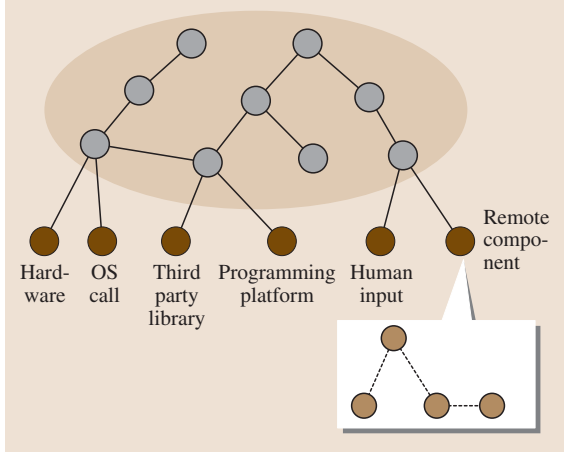

Fig. 24.23 Identification of atomic components in an SoS

Verify the legitimacy of results;
Handle exceptions;
If results fail, report a failure;
}

The data collector performs acceptance testing on the inputs and outputs of the atomic component and maintains following data in a local log file:

- The execution counter keeps track of how many times the atomic component has been called.
- The failure counter counts how many failures have been detected.
- Failure types: incorrect data, exceptions, and crash, etc.

The local log file associated with the data collector automatically synchronizes with a central database that records data related to all atomic components. The synchronization can be performed in the testing stage, in the development process, and during the operational stage after the software is delivered to the client. Online error reporting during the operational stage can also help the developers to design patches and updates of the software. The execution number of each atomic component collected during the operational stage can be used to determine the execution profile of the software.

For each atomic component, the following data items are maintained in the database:

- Versions: recording the date and time of each modification/error correction performed on the atomic component. Each error correction results in a new version of the component;
- Number of executions between two error corrections;
- Number of failures between two error corrections;
- Numbers of each failure type.

Table 24.6 Reliability definition of ACDATE entities

ACDATE	Reliability of the ACDATE entity
	The *probability* that the:
Actor	– actor presents the expected behavior
Condition	– condition presents the expected Boolean value
Data	– data presents the expected value
Action	– action presents the expected behavior
Timing	– action completes in given time frame
Event	– event is sent or received successfully
Scenario	– scenario presents the expected behavior
System	– system presents the expected behavior

These data can be used to estimate the reliability of the components. In the next subsection, we will apply the input domain-based reliability growth model to estimate the reliability of each atomic component.

An incorrect output of a program is a failure. A program contains errors if it can produce a failure when certain input cases are applied. The size of an error is the ratio of the number of inputs that can detect the error (cause a failure) and the total number of valid inputs.

According to the input domain-based reliability growth model [24.38, 39], the failure data stored in the central database can be used to estimate the error sizes $\Theta_1, \Theta_2, \ldots, \Theta_k$ and the failure rates $\lambda_1, \lambda_2, \ldots, \lambda_n$ of the errors in each atomic component between two error corrections. The error sizes and failure rates between error corrections can be used to estimate the final failure rate λ of each atomic component.

The failure rate and the total number of executions associated with each atomic component is the input to the structural reliability model to be discussed in the remaining part of the paper.

24.6.2 The Architecture-Based Reliability Model

In the previous subsection, we evaluated the reliability of atomic components. In this subsection, we evaluate the reliability of an SoS consisting of multiple systems, each of which is considered an atomic component. The model can be generalized to evaluate a system or a subsystem in a system, with the knowledge of the reliability of its components, operational profile, and the architecture of the system. The architecture determines the contribution of the reliability of each atomic component to that of the overall system. Hence the approach is named the architecture-based reliability model [24.40].

First we give the definition of a component's reliability and present our assumptions; then we discuss how the architecture affects the propagation of reliability; finally, we derive the formulas that compute the reliability.

We base our reliability model on the ACDATE scenario model, which describes the structure of a system using model entities *actors*, *conditions*, *data*, *actions*, *timing*, and *events*, and the behavior of a system using *scenarios*. The ACDATE scenario model models the general computing process.

The *reliability* definitions of ACDATE entities and scenarios are summarized in Table 24.6.

The assumptions of our reliability are

1. Assignment assumption: the assignment operation introduces no new failure.
2. Condition assumption: the condition fails when any data that constitutes the condition fails.
3. Acyclic dependency assumption: there is no cyclic dependency among ACDATE entities.

The system behaviors are specified by a few system-level scenarios and the reliabilities of those system-level scenarios will contribute to that of the system. Different scenarios may have different execution rates (the operational profile), which determine the weight of the contributions.

A scenario is a sequence of activities connected by four operators: *sequence*, *choice*, *loop* and *concurrency*. Each *activity* is a data assignment, exchanging an event, doing an action, or executing a sub-scenario. Hence the reliability of data, events, actions and sub-scenarios will contribute to that of a scenario. The choice and loop operator are associated with one or more conditions that determine the branches to take. Hence the reliability of conditions also contributes to that of a scenario. Moreover, the *true/false* rate of each condition will affect the reliability of the scenario through the choice and loop operators (formulas will be presented later).

Each top-level scenario would be invoked by an *external* event. Hence, the occurrence rates of external events determine the operational profile of top-level scenarios. A scenario may emit an *internal* event, whose sole function is to resume or invoke the execution of a sub-scenario. Hence, the occurrence rates of internal events will affect the operational profile of sub-scenarios (in addition to direct calling from other scenarios). The occurrence rates of internal events can be determined by that of external events invoking top-level scenarios, and the control flow of those scenarios that emit internal events.

To summarize, assuming that we know the reliability of each scenario and the occurrence rate of each external event (and hence internal events), we can evaluate the reliability of the system following the formula:

$$Rel_{\text{system}} = \left[\sum (w_i * Rel_{\text{scenario\_}i})\right] / \sum (w_i) ,$$

where w_i is the execution rate of the corresponding scenario. In the following we present the calculation of the reliability of a scenario.

The reliability of data is determined by that of its storage method, which is modeled as atomic components (memory, external database, file system, etc.), and hence is known. The reliability of actions is known if it is atomic (e.g., a system call), or is that of the sub-scenario that implements it. The reliability of events is determined by that of the communication link (atomic component) and hence is known.

Following assumption 1, the reliability of assignment is that of the right-hand-side data. Hence we know the reliability of each activity in a scenario.

If several activities are connected by a sequence operator, then the overall reliability follows the formula:

$$Rel_{\text{sequence}} = \prod Rel_{\text{activity\_}i} ,$$

where $Rel_{\text{activity\_}i}$ is the reliability of each activity that participates in the sequence. If several activities are connected by a concurrency operator and all of them are replicas, then the overall reliability follows the formula:

$$Rel_{\text{concurrency}} = 1 - \prod (1 - Rel_{\text{activity\_}i}) .$$

Otherwise, it is the same as the sequence, since any failure results in the failure of the overall concurrency. For the loop operator, the formula is:

$$Rel_{\text{loop}} = (Rel_{\text{cond\_set}} \cdot Rel_{\text{block}})^{Pt} ,$$

where $Rel_{\text{cond\_set}}$ is the reliability of the condition set associated with the loop operator, Rel_{block} is the reliability of the block of activities enclosed in the loop operator, and Pt is the expected number of loops. We will discuss $Rel_{\text{cond\_set}}$ later. For the choice operator, the formula is

$$Rel_{\text{choice}} = Rel_{\text{cond}_{\text{set}}} \cdot \left(Pt \cdot Rel_{\text{true\_block}} + (1 - Pt) \cdot Rel_{\text{false\_block}}\right) ,$$

where Pt is the probability that the condition set evaluates as *true*. Since a scenario consists of only these four types of operators, we can calculate its reliability following these formulas. The reliability of a condition is determined by the data that constitute the condition, or is known if the condition is atomic (e.g., a system call). Following assumption 2, if a condition consists of several data, its reliability is the product of the reliabilities of all the data. We omit the deduction process due to the space restriction and only present the final formulas here:

$$Rel_{\text{cond\_set}} = 1 - \sum ProTop_{c(m,o)} ,$$

$$\text{where each } ProTop_{c(m,o)} = \sum ProTop_{\{c\}o}$$

Table 24.7 The most reliable services and their forecast

Components	Reliability	Forecast probability	Adjusted probability
RainForecast	0.764	18 % heavy rain	33.1%
TempForecast	0.98	31 % extreme temp	31.76 %
WindForecast	0.90	23 % strong wind	28.4 %

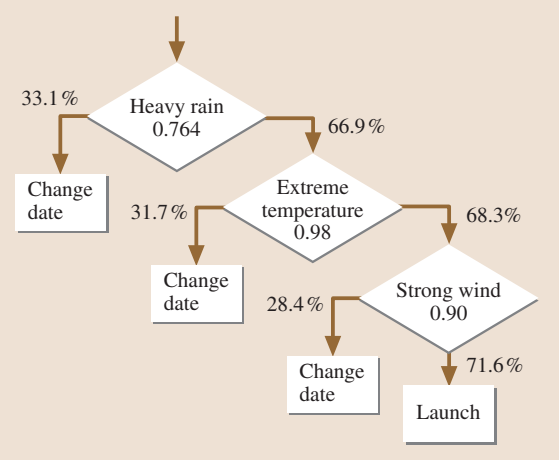

Fig. 24.24 Decision making

Each $ProTop_{\{c\}o}$ is calculated by the following formulas:

$$ProTop_{\{c\}o} = Rel_{\{c\}o} \cdot \left(ProTF_{\{c\}o} + ProFT_{\{c\}o}\right) \cdot Prob_{remv} \,,$$

$$Rel_{\{c\}o} = \prod(1 - Rel_{Ck}) \cdot \prod(Rel_{Cl}) \,,$$

where Rel_{Ci} is the reliability of the i-th condition in the condition set. Each condition set is evaluated in disjunction normal form (DNF), whose Boolean value may be dominated by a true disjunct. $Prob_{remv}$ is used to compensate for possible domination and is defined as the probability that the disjunct evaluates to be false. $ProTF_{\{c\}o}$ and $ProFT_{\{c\}o}$ are the probabilities that a condition incorrectly changes from *true* to *false*, or from *false* to *true*, respectively, and are determined by the condition's reliability.

24.6.3 Applications

This subsection uses an example to illustrate the applications of the proposed dynamic software reliability model. Assume a space agency plans to launch a satellite on a specific date and from a specific location. Among other constraints, the launch is heavily dependent on the weather conditions at the launch location, including rain, wind, and temperature. Three independent weather services are used, offering RainForecast, TempForecast, and WindForecast, respectively. The forecasts are given with their probabilities. The reliabilities based on the history of the services, their forecast probabilities (component outputs), and the adjusted probabilities based on the reliability and the forecast probabilities are given in Table 24.7.

To decide whether to change the launch date based on the weather forecasting information, the space agency then constructed a system based on the components. The reliability of the system and the final decision of whether to launch the satellite can then be assessed by the process shown in Fig. 24.24. The numbers in the diamond boxes are the reliabilities of each component. The numbers on the branches are the probabilities forecasted by the best service. The decision is based on these two factors.

24.6.4 Design-of-Experiment Analysis

Design of experiment (DOE) is an engineering technique [24.41] that can be used to determine the extent of the impact of the parameters (factors) of a model on the final results. This subsection applies DOE to analyze the impact of the reliability of the components on the reliability of the SoS.

There are three factors in the example, the reliabilities of (A) RainForecast, (B) TempForecast, and (C) WindForecast. We use two-level DOE techniques, i. e., we use high and low values of each factor: RainForecast (70%, 90%), TempForecast (90%, 99%) and WindForecast (85%, 95%). In our experiment, the three-factor and two-level design generated the analysis of variance (ANOVA) table shown in Table 24.8.

The F-value represents the significance of the impact of a model and its components. In general, if a com-

Table 24.8 ANOVA significance analysis

Source	F-value	Prob > F-value
Model	$1.421 \times 10^{+5}$	< 0.0001
A (RainForecast)	$3.898 \times 10^{+5}$	< 0.0001
B (TempForecast)	$2.2943 \times 10^{+4}$	< 0.0001
C (WindForecast)	$1.3558 \times 10^{+4}$	< 0.0001

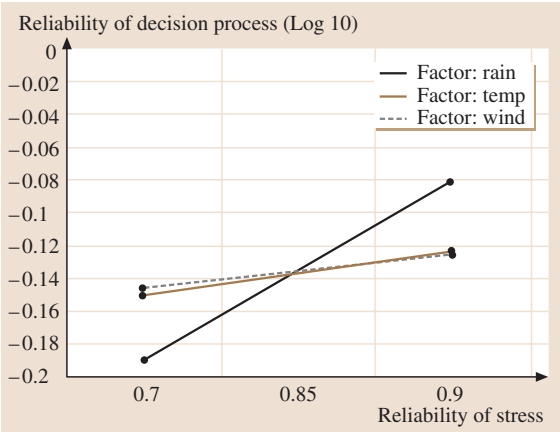

Fig. 24.25 Impact of components

ponet generates a significance value (Prob > F-Value) of less than 0.05, the impact of the component is significant. For example, the F-value $1.421 \times 10^{+5}$ for the overall model in the table implies that the model is significant. There is only a 0.01% chance that the an F-value of this size could occur for this model due to noise.

The experimental results in Table 24.8 also show that the F-values and significances of RainForecast, TempForecast, and WindForecast are all less than 0.0001, and thus they are all significant model components.

DOE can be used to compare the significances among the components. Figure 24.25 shows the impacts of the three components on the overall reliability in our example. As can be seen, the higher the component reliability, the higher the overall reliability. However, the impact of the RainForecast service is much more significant than that of the others. This suggests that the space agency should pay more attention to the quality of the rain-forecast service provider.

24.7 The Fourth Generation of E2E T&E on Service-Oriented Architecture

Service-oriented architecture (SOA) and web services (WS) are emerging technologies that may change the way computer software is designed and used. Many industrial standards have been defined in the past few years to facilitate and regulate the development of WS. However, there are still a number of barriers preventing WS from being widely applied or being used as the platform for trustworthy and high-assurance systems. *Sleeper* identified five missing pieces of WS technology: reliability, security, orchestration, legacy support, and semantics [24.42]. Among these five issues, reliability is the least addressed and probably most difficult, for the following reasons:

- WS are based on an unreliable and open internet infrastructure, yet they are expected to be trustworthy.
- WS have a loosely coupled architecture, yet they are expected to collaborate closely and seamlessly.
- WS can be invoked by unknown parties with unpredictable requests, and thus WS must be robust.
- WS involve runtime discovery, dynamic binding with multiple parties, including middleware and other WS, and runtime composition using existing WS. Thus, WS must support dynamic and runtime behaviors.
- WS must support dynamic configuration and reconfiguration to support fault-tolerant computing.
- WS must support dynamic composition and recomposition to cope with the changing environment and changing requirements.
- WS involve concurrent threads and object sharing. It is difficult to test concurrent processes.

We propose an integrated collaborative and cooperative WS development process to achieve high-assurance computing. The process is implemented in a framework consisting of three major modules dealing with the construction, publishing, and testing of WS.

As shown in Fig. 24.26, the WS cooperative and collaborative computing (WSC3) framework consists of three modules: cooperative WS construction; publishing; and testing, assessment, and ranking (WebStrar). The framework can be used by service requestors, service providers, as well as researchers experimenting with WS.

As an example, the arrows and numbers in Fig. 24.26, which outline cooperation scenarios between the components, are explained as follow.

1. The WS construction module, in the process of dynamically constructing a composite WS based on existing WS, requests information from the WS publishing module.
2. The publishing module provides required information, including the specification and interface of using the WS.

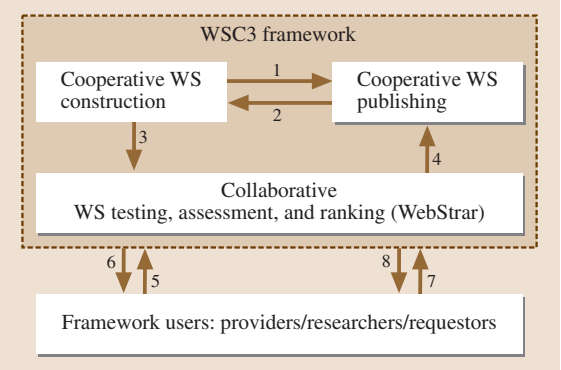

Fig. 24.26 WS cooperative and collaborative computing framework

3. After a composite WS is constructed, the construction module submits the WS to the WebStrar module for rigorous testing.
4. If the WS passes the test, it will be registered with the publishing module and a new WS is available for online access.
5. A WS provider or a researcher submits their WS for publication or testing. The WS will be tested by the WebStrar module rigorously based on the test scripts submitted by the provider as well as the test scripts generated by WebStrar. Sharing the test scripts represents collaboration between the framework and WS providers and researchers.
6. The framework publishes the WS and informs the WS provider if the submitted WS passes the test.
7. A WS requestor requests a service. The requestor can request testing before using a WS. It can use the test scripts provided by the framework or submit their own test scripts. Sharing the test scripts represents collaboration between the framework and WS requestors. WS requestors can also access the reliability data and ranking information of published WS.
8. The framework processes and responds to the WS requestor.

In the following three subsections, we elaborate the three modules in the WSC3 framework, respectively.

24.7.1 Cooperative WS Construction

Figure 24.27 elaborates the cooperative WS construction module in Fig. 24.26. This module has six components.

The cooperative WS specification component provides guidelines and tools for users to write WS specifications in a specification language, e.g., in OWL-S.

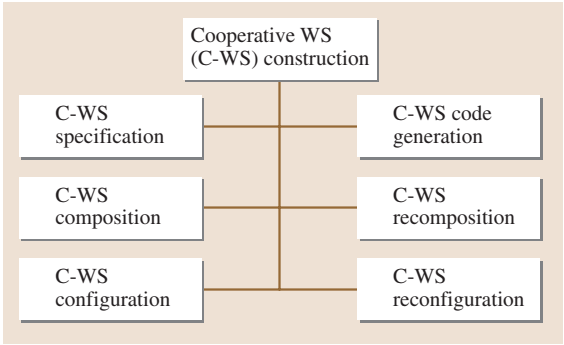

Fig. 24.27 Cooperative WS construction

The component will then use WebStrar to perform a consistency and completeness (C&C) check on the specification.

Once the specification passes the check, the code-generation component can automatically generate the executable code. The WS generated in this way is atomic, because its implementation detail is not accessible to the users. All WS submitted by WS providers are also atomic.

The cooperative WS composition component provides automated high-level WS composition based on existing atomic WS and their specifications.

The recomposition component can reconstruct a composite WS if the requirement and specification are changed. Composition and recomposition components construct WS based on the functional requirement while the configuration and reconfiguration components deal with the management of redundant resources and the reliability of WS. The configuration component adds redundant structure into composite WS to meet the reliability requirements while the reconfiguration component maintains redundancy after the environment is changed, for example, if some WS become faulty or unavailable.

24.7.2 Cooperative WS Publishing and Ontology

This section elaborates the cooperative WS publishing module in Fig. 24.26. Current WS publishing is based on the universal description, discovery, and integration (UDDI) technique. The UDDI discovery part is based on simple term/text matching, which does not have the intelligent to find synonyms and semantically related terms. For example, if the phrase "red wine" is searched, terms like Cabernet Sauvignon and Merlot should be found too.

Table 24.9 Cooperative versus traditional ontology

	Traditional ontology	Cooperative ontology
Test scripts	Does not include test scripts	Include test scripts and execute these test scripts at runtime
Nonfunctional property	Use certain terms to represent the value of these nonfunctional properties, such as performance and security.	Use test scripts to present the nonfunctional properties. Translate the nonfunctional properties to the measurable features.
Behavior constraints	Allow specification of the constraints, but not their execution at runtime	Execute the constraints at runtime to check if the assigned services match the constraints.
Interface	Does not include the interface information in the description	Use specific test scripts to present the interface constraints

OWL-S and Protégé are recent projects that support ontology description. Ontology defines the basic terms and relations comprising the vocabulary of a topic area as well as the rules for combining terms and relations to define extensions to the vocabulary [24.43]. This represents knowledge about a domain and describes specific situations in the domain [24.44]. Dynamic composition and recomposition need to discover WS at runtime over the internet, add them to the ontology domain, and then compose a WS at runtime.

Current ontology methods do not include the verification process. In our cooperative WS publishing module, we integrated the collaborative verification and validation (CV&V) process into the ontology by including the necessary test scripts in the ontology domain. When a WS is chosen for composition, recomposition, configuration, or reconfiguration, the stored test scripts will be immediately applied to test the WS. This integrated ontology with CV&V is called cooperative ontology. Table 24.9 compares and contrasts cooperative and traditional ontology.

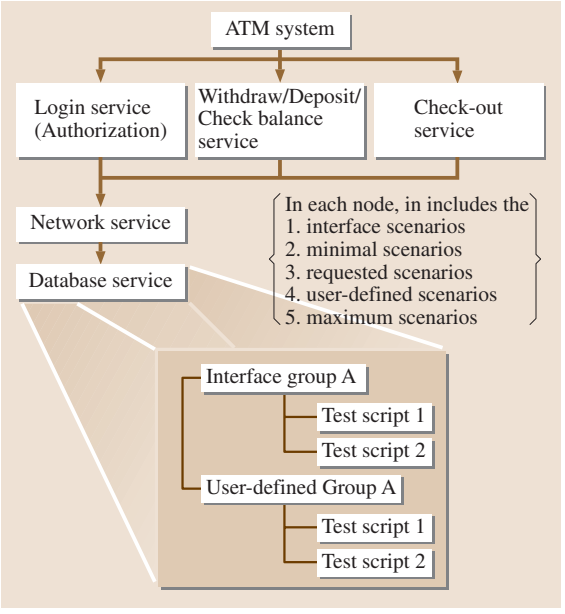

Fig. 24.28 ATM services tree example

The ontology-based architecture plays a key role in runtime WS composition. Runtime verification can choose different levels of test scripts to verify the services found. To further explain the idea, a simple automatic teller machine (ATM) example is used here. Assume the ATM offers login, balance-checking, withdrawal, deposit, and logout services.

The service tree of the cooperative ontology representing the ATM composite WS is given in Fig. 24.28. Each node has a service interface definition, a number of service constraints and service scenarios, and user-specific requirements described using test scripts. For instance, if we want to choose a login service that supports a specific character set [&*%] in the user name, the user-defined test script can be:

Execute Register("abc&^%123", "123456");*

If this test script fails, the services found do not conform to the requirement and will be rejected. As discussed before, there are multiple levels of test scripts, which include the interface test scripts. In the service tree specification, the internal relations among test scripts are also important to support dynamic service composition and recomposition.

24.7.3 Collaborative Testing and Evaluation

The WS composed using the process in Fig. 24.29 will be tested, assessed and ranked. Figure 24.4 depicts the module that tests and assesses the reliability of the WS and assures the tools involved. The solid arrows indicate that a component can be decomposed into several subcomponents, while the dotted arrows indicate the data flow between components. WebStrar itself is a framework supporting the development of trustworthy WS

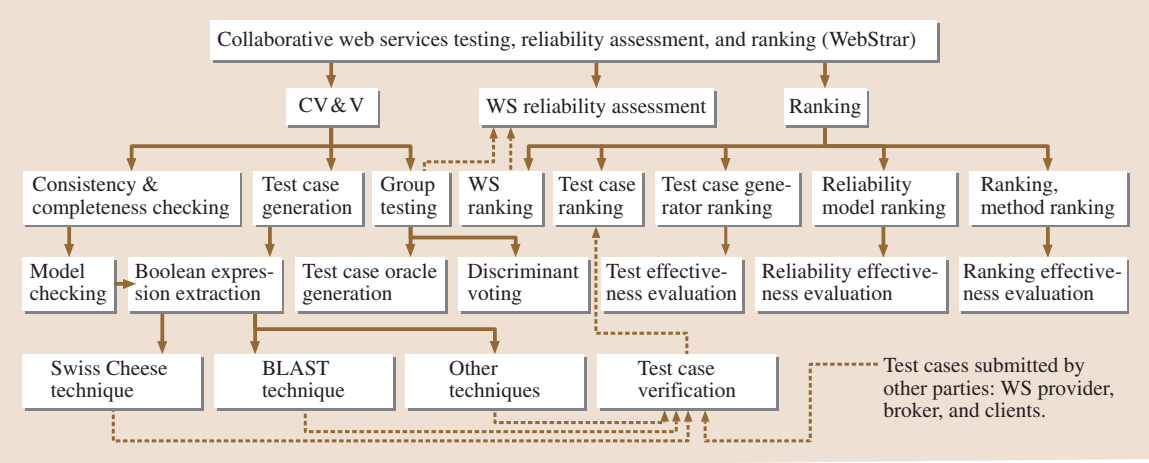

Fig. 24.29 Collaborative testing, assessing, and ranking

(http://asusrl.eas.asu.edu/srlab/projects/webstrar/index.htm). This section explains the individual techniques developed and to be developed in this framework. At the top level, WebStrar consists of three components: CV&V, WS reliability assessment, and ranking.

The idea of CV&V is to involve all parties (WS providers, brokers, and clients) in verifying and validating the WS, because the WS provider does not have full information on how their WS will be used by clients and brokers in user-composed composite WS [24.45, 46]. Before test-script generation, the consistency and completeness of the WS specification in OWL-S or web services description language (WSDL) will be checked through model checking or other methods, which may detect inconsistent conditions or incomplete coverage of the requirements [24.17]. Once the specification passes this check, Boolean expressions can be extracted, which can be used for test-script generation. Different techniques can be applied here; we have applied the Swiss cheese [24.17] and BLAST [24.16] techniques in our experiments. The system-generated test scripts, along with the test scripts provided by other parties, will be verified for correctness and ranked according to their effectiveness at detecting faults in WS.

Group testing is a key technique developed to test the potentially large numbers of WS available on the internet [24.46]. WS with the same specification can be tested in groups and the results are compared by a discriminant voter, which can identify correct and faulty output based on the majority principle. The majority results are used as the oracles for future testing.

Reliability assessment of WS is different from that of traditional software. WS reliability models do not have access to the WS source code. WS reliability can only be assessed at runtime because WS can be composed and modified (recomposed) at runtime. A group-testing-based dynamic reliability model has been developed to assess WS reliability [24.40]. Reliability is one of the criteria used to rank WS. Other criteria include security, performance, real-time ability, etc.

WebStrar allows researchers and WS providers to submit their models and tools for evaluation and ranking. WebStrar supports ranking of test scripts in terms of fault-detection capacity, test-script generation algorithms in terms of generation of effective test scripts, reliability models in terms of the accuracy of their assessment results, and ranking models themselves in terms of ranking accuracy.

24.8 Conclusion and Summary

E2E T&E technology was initially developed for DoD command-and-control systems and later applied in various industrial projects. It was initially designed as an integrated testing technology and later developed

into a full development technology spanning requirements; specification; model checking; code generation; test-case generation; testing; simulation; policy specification and enforcement; and reliability, security, and risk enforcement and assessment. All these techniques are coherently based on the scenario specification. From the software architecture point of view, E2E T&E technology has evolved from centralized architecture, through distributed agent architecture, to service-oriented architecture.

E2E T&E technology has been successfully applied in several DoD command-and-control projects where high-assurance computing is required and in several civilian projects including embedded systems in business networks. The application of the technology and its tools dramatically reduce the development cycle of the end systems and increase their dependability.

Systems developed using E2E T&E technology have the following attributes and features.

The developed systems are flexible and can adapt to changing environments. Some important attributes of *adaptability* include *speed*, *scalability*, *reusability*, *partitioning*, and *integration*. In general, the system is adaptive as it supports rapid development; is scalable from small applications to large applications; has many reusable tools that can produce reusable components; and has an integrated process.

The E2E T&E development process is fast because it includes tools that perform all jobs automatically where possible. It generates test cases automatically from the system specification and policy specification. It generates an executable automatically, and performs distributed test execution automatically. The evaluation process is also automatic.

The E2E T&E process is scalable because it can apply to large as well as small applications. The scenarios used in the E2E process are hierarchical and thus can apply to the hierarchical structure of a large SoS or a small subsystem.

The E2E T&E process has many reusable tools, including the scenario specification tool, the test-case management tool, the scenario simulation tool, and the distributed test-execution tool. One of the key benefits of the E2E tool is that specified scenarios are highly reusable and can be easily changed. System scenarios keep on changing as new requirements become known and new technology is introduced during system development; changing system scenarios with the E2E tool is much easier than redeveloping scenarios by hand. In most cases, new scenarios are developed by changing existing scenarios, and changing scenarios using the E2E tool with dependency analysis is easier than starting from scratch without any tool support. Furthermore, once new scenarios are specified, they can be automatically analyzed by various techniques such as timing analysis, and new test scripts can be rapidly generated and executed.

E2E T&E tightly integrates system analysis and modeling with integration testing because the same techniques, i. e., scenarios, can be used for both system analysis as well as integration testing. The importance of testing has recently being emphasized by agile development processes such as extreme programming. While testing is one of their main techniques, agile processes do not have such tight integration between system analysis and testing as the DoD E2E T&E. Tight integration changes the way systems are developed; instead of performing *requirement-driven testing* only, the E2E process calls for *test-based requirement analysis*. In other words, the requirements should be developed in a way that can be used for rapid integration testing (by automated test-script generation, verification patterns, and distributed test execution) and evaluation (by various analyses and simulation). In fact, E2E T&E supports a test-based development process from requirements to operation and maintenance, and such a process is compatible with agile development processes or incremental development.

References

24.1 R. S. Pressman: *Software Engineering: A Practitioner's Approach*, 5th edn. (McGraw Hill, New York 2000)

24.2 S. Kirani, W. T. Tsai: *Specification and Verification of Object-Oriented Programs*, tech. rep., Department of Computer Science and Engineering (Univ. Minnesota, Minneapolis 1994)

24.3 D. C. Kung, P. Hsia, J. Gao: *Testing Object-Oriented Software* (IEEE Computer Society, Los Alamitos, CA 1999)

24.4 W. T. Tsai, Y. Tu, W. Shao, E. Ebner: Testing extensible design patterns in object-oriented frameworks through hierarchical scenario templates, Proc. COMPSAC **23**, 166–171 (1999)

24.5 W. T. Tsai, V. Agarwal, B. Huang, R. Paul: Augmenting Sequence Constraints in Z and its Application to Testing. In: *Proc. 3rd IEEE Symposium on Application-Specific Systems and Software Engineering Technology* (IEEE Computer Society Press, Los Alamitos 2000) pp. 41–48

24.6 DoD OASD C3I Investment, Acquisition: *End-to-End Integration Testing Guidebook* (IEEE Computer Society Press, Los Alamitos 2001)

24.7 W. T. Tsai, X. Bai, R. Paul, L. Yu: *Scenario-Based Functional Regression Testing* (IEEE Proc. of COMPSAC, Chicago 2001) pp. 496–501

24.8 W. T. Tsai, C. Fan, R. Paul, L. Yu: *Automated Event Tree Analysis Based-on Scenario Specifications* (Proc. of IEEE ISSRE, IEEE Computer Society Press, Los Alamitos 2003) pp. 240–241

24.9 W. T. Tsai, X. Bai, R. J. Paul , W. Shao, V. Agarwal: *End-To-End Integration Testing Design* (Proc. of IEEE COMPSAC, Chicago 2001) pp. 166–171

24.10 X. Bai, W. T. Tsai, R. Paul, K. Feng, L. Yu: *Scenario-Based Modeling and Its Applications to Object-Oriented Analysis, Design, and Testing*, Proc. of IEEE WORDS 2002 (IEEE Computer Society Press, Los Alamitos 2002) pp. 140–151

24.11 W. T. Tsai, F. Zhu, L. Yu, R. J. Paul: *Rapid Verification of Embedded Systems Using Patterns* (COMPSAC, IEEE Computer Society Press, Los Alamitos 2003) pp. 466–471

24.12 F. Zhu: A Requirement Verification Framework for Real-Time Embedded Systems. Ph.D. Thesis (Dept. of Computer Sci. and Engineering, Univ. of Minnesota, Minneapolis 2002)

24.13 A. Cockburn: *Agile Software Development* (Addison Wesley, Reading, MA 2001)

24.14 W. T. Tsai, R. Paul, L. Yu, A. Saimi, Z. Cao: Scenario-Based Web Service Testing with Distributed Agents, IEICE Trans. Inf. Syst. **E86-D**(10), 2130–2144 (2003)

24.15 W. T. Tsai, A. Saimi, L. Yu, R. Paul: *Scenario-Based Object-Oriented Test Frameworks* (Proc. Third Int. Conf. Quality Software (QSIC03), IEEE Computer Society Press, Los Alamitos 2003) pp. 410–417

24.16 D. Beyer, A. Chlipala, T. Henzinger, R. Jhala, R. Majumdar: Generating Tests from Counterexamples, Proc. 26th Int. Conf. Software Engineering (ICSE'04), Scotland, UK May 2004 (IEEE Computer Society, Washington, DC 2004) 326–335

24.17 W. T. Tsai, X. Wei, Y. Chen, B. Xiao, R. Paul, H. Huang: *Developing and Assuring Trustworthy Web Services* (7th International Symposium on Autonomous Decentralized Systems (ISADS), Chengdu, China, IEEE Computer Society Press, Los Alamitos April 2005)

24.18 F. Zhu: A Requirement Verification Framework for Real-Time Embedded Systems. Ph.D. Thesis (Department of Computer Science and Engineering, Univ. of Minnesota, Minneapolis 2002)

24.19 R. M. Fujimoto: Parallel and Distributed Simulation. In: *Proc. 1999 Winter Simul. Conf.*, ed. by P. A. Farrington, H. B. Nembhard, D. T. Sturrock, G. W. Evans (ACM Press, New York 1999) pp. 122–131

24.20 W. T. Tsai, W. Song, R. Paul, Z. Cao, H. Huang: *Services-Oriented Dynamic Reconfiguration Framework for Dependable Distributed Computing* (COMPSAC, IEEE Computer Society Press, Los Alamitos Sep. 2004) pp. 554–559

24.21 W. T. Tsai, Y. Chen, Z. Cao, X. Bai, H. Huang, R. Paul: *Testing Web Services Using Progressive Group Testing*, Advanced Workshop on Content Computing (Lecture Notes in Computer Science 3309, Springer Berlin, Zhenjiang 2004) pp. 314–322

24.22 E. Clarke, O. Grumberg, D. Peled: *Model Checking* (MIT Press, Cambridge, Massachusetts 2002)

24.23 T. Y. Chen, M. F. Lau: *Test Cases Selection Strategies Based on Boolean Specifications*, Software Testing, Verification and Reliability, Vol. 11 (Wiley, Atrium, UK Sep. 2001) pp. 165–180

24.24 IEEE Std1516-2000: IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Framework and Rules (2000)

24.25 J. Davila, E. Gomez, K. Laffaille, K. Tucci, M. Uzcategui: MultiAgent Distributed Simulation with GALATEA, 9th IEEE International Symposium on Distributed Simulation and Real-Time Applications 2005 (IEEE Computer Society Press, Los Alamitos 2005) 165–170

24.26 L. F. Wilson, D. J. Burroughs, A. Kumar: A framework for linking distributed simulations using software agents, Proc. IEEE **89**(2), 135–142 (Feb 2001)

24.27 B. Logan, G. Theodoropoulos: The distributed simulation of multi-agent systems, Proc. IEEE **89**(2), 174–185 (2001)

24.28 H. S. Sarjoughian, B. P. Zeigler, S. B. Hall: A Layered Modeling and Simulation Architecture for Agent-Based System Development, Proc. IEEE **89**(2), 201–213 (2001)

24.29 C. Pleeger: *Security in Computing*, 3rd edn. (Prentice Hall PTR, Indianapolis 2000)

24.30 N. Damianou, A. Bandara, M. Sloman, E. Lupu: *A Survey of Policy Specification Approaches*, Technical Report (Dept. of Computing, Imperial College of Sci. Technology and Medicine, London, UK 2002)

24.31 M. Kangasluoma: *Policy Specification Languages*, Technical Report (Dept. of Computer Science, Helsinki Univ. of Technology, Helsinki Nov. 1999)

24.32 N. Damianou, N. Dulay, E. Lupu, M. Sloman: *The Ponder Policy Specification Language*, Proceedings of Workshop on Policies for Distributed Systems and Networks, 2001

24.33 L. Kagal, Rei: *A Policy Language for the Me-Centric Project*, Technical Report (HP Laboratories, Palo Alto, CA 2002)

24.34 D. Bell, L. LaPadula: *Secure Computer System: Unified Exposition and Multics Interpretation*, Technical Report (MITRE Corporation, Bedford, MA March 1976)

24.35 R. Sandhu, E. Coyne, H. Feinstein, C. Youman: Role-based access control models, IEEE Comput. **29**(2), 38–47 (1996)
24.36 E. Bertino: RBAC models – concepts and trends, Comput. Security **22**(6), 511–514 (2003)
24.37 S. Osborn, R. Sandhu, Q. Nunawer: Configuring role-based access control to enforce mandatory and discretionary access control policies, ACM Trans. Inf. Syst. Security **3**(2), 85–106 (2000)
24.38 Y. Chen: Modeling Software Operational Reliability under Partition Testing. In: *IEEE 28th Annual International Symposium on Fault-Tolerant Computing (FTCS-28), Munich, June 1998* (IEEE Computer Society Press, Los Alamitos, CA 1998) pp. 314–323
24.39 Y. Chen, J. Arlat: An Input Domain-Based Software Reliability Growth Model under Partition Testing with Fault Corrections, Proc. 15th Int. Conf. Computer Safety, Reliability, Security (SAFECOMP'96), Vienna October 1996 (Springer, Berlin 1997) 136–145
24.40 W. T. Tsai, D. Zhang, Y. Chen, H. Huang, R. Paul, N. Liao: A Software Reliability Model for Web Services. In: *8th IASTED Int. Conf. Software Eng. Appl., Cambridge MA, November 2004*, ed. by M. H. Hamza (ACTA Press, Calgary, Canada 2004)
24.41 D. C. Montgomery: *Design and Analysis of Experiments*, 5th edn. (Wiley, Indianapolis 2000)
24.42 B. Sleeper: The five missing pieces of SOA. www.infoworld.com/article/04/09/10/37FEwebservmiddle_1.html (2004)
24.43 R. Neches, R. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator, W. R. Swartout: Enabling technology for knowledge sharing, AI Magazine **12**(3), 36–56 (1991)
24.44 R. Fikes, A. Farquhar: Distributed repositories of highly expressive reusable ontologies, IEEE Intell. Syst. **14**(2), 74–79 (1999)
24.45 W. T. Tsai, R. Paul, Z. Cao, L. Yu, A. Saimi, B. Xiao: Verification of web services using an enhanced UDDI server, Proc. IEEE WORDS , 131–138 (2003)
24.46 W. T. Tsai, Y. Chen, R. Paul, N. Liao, H. Huang: Cooperative and group testing in verification of dynamic composite web services. In: *Workshop on Quality Assurance and Testing of Web-Based Applications, in conjunction with COMPSAC* (IEEE Computer Society Press, Los Alamitos, CA Sep. 2004) pp. 170–173