MASTER COPY KEEP THIS COPY FOR REPRODUCTION PURPOSES

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	October 23, 1995	Phase I STTR Final Report 9/94-9/95

4. TITLE AND SUBTITLE
Integrated Environment for Performance, Reliability and Availability Modeling

5. FUNDING NUMBERS

6. AUTHOR(S)
Alex Blakemore, Genoa Software Systems
Gianfranco Ciardo, College of William and Mary

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Genoa Software Systems
10811 Midsummer Drive, Reston, VA 22091-5116
College of William and Mary, Dept of Computer Science
P. O. Box 8795 Williamsburg, VA 23187-8795

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U. S. Army Research Office
P. O. Box 12211
Research Triangle Park, NC 27709-2211

11. SUPPLEMENTARY NOTES
The view, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
No limitation upon distribution of report, as long as original authorship is acknowledged.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

There are a variety of analysis techniques and representation formalisms that have been developed to support the performance, reliability and availabilty analysis of complex systems. Such techniques include discrete-event simulation, numerical analysis of stochastic processes and product-form combinatorial methods. All these techniques have different strengths and limitations. The research community has developed several new specification and analysis techniques that have demonstrated dramatic improvements in capability over the technology in common use today.

Most of these new techniques exploit some form of system decomposition, requiring the combination of multiple submodels, perhaps represented using different formalisms, and possibly analyzed in parallel using different computers. This STTR project aimed to design a new architecture for an integrated mdoeleing environment that would support the fundamental operations necessary to allow sophisticated modeling techniques to be applied easily in practice. The project led to the development of the SMART and MDL languages, and a corresponding academic prototype.

14. SUBJECT TERMS	15. NUMBER OF PAGES
performance, reliability, availability, modeling, stochastic processes, simulation, analysis	37, incl. appendix
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

Integrated Reliability and Performance Modeling Environment

ARO STTR Phase I Final Report

October 1995

Genoa Software Systems
College of William and Mary

Project Summary

The Army Small Business Technology Transfer (STTR) project, *Integrated Reliability and Performance Modeling Environment,* examined the feasibility of producing a commercial software environment for performance and reliability modeling that supports new techniques for analyzing large complex systems.

The project was a joint effort between Genoa Software Systems and the Computer Science Department of the College of William and Mary sponsored by the Army Research Office (ARO). Dr. Kishor Trivedi of the Electrical Engineering Department at Duke University provided guidance as a consultant. William and Mary also received partial matching funds from the VA Center for Innovative Technology (CIT).

The project focused on exploring possible software architectures for a modeling environment that facilitates the use of both recent and classical advances in the specification and analysis of complex stochastic models. During Phase I progress, a system architecture was developed and refined, and a working prototype of the envisioned environment was constructed.

This report describes the project scope and goals, summarizes the technical results, and discusses commercialization issues.

I. Project Scope and Goals

The ultimate goal of this project was to close the gap between the state-of-research and state-of-practice in modeling by weaving new techniques available from academia into a toolset that would significantly increase the ability to analyze large, complex systems. We proposed to develop a toolset that supports modeling as an integral part of the design process, shields the user from complex modeling languages, and takes advantage of the most appropriate formalism(s) and state-of-the-art solution methods. The end-product was intended to integrate many of the recent advances for modeling into a set of open-architecture tools suitable for commercial use.

The primary *technical objectives* for phase I were to develop a viable toolset design and architecture, validate the design using a prototype implementation, and experiment with the practical application of new ideas (e.g., parallelism and decomposition.)

The primary *commercial objectives* for phase I were to select an initial target marketplace, develop a viable plan for marketing, and ensure the product matches the chosen market and to identify potential commercialization partners.

Background and Motivation

Different forms of stochastic models are useful for quantitatively analyzing complex systems in several engineering disciplines including computer architecture, communication networks, and aerospace engineering. The chief difficulty with current modeling techniques is that they often require immense computational resources in order to analyze even moderately complex systems. Because of these limitations, accurate analysis of large complex systems is often costly and frequently impractical.

Researchers have repeatedly demonstrated dramatic improvements in the capacity to analyze large and complex systems using techniques such as time-scale and hierarchical decomposition, hybrid models and iterative solution methods. Unfortunately, such techniques are often difficult, if not impossible, to apply with most commercially available modeling tools due to their monolithic restrictive architectures.

Two concepts pervade the design of any modeling toolset: formalisms and solution methods. A *formalism* is a language in which models can be precisely expressed. Once specified using an appropriate formalism, models can be analyzed using one of many available *solution methods* in order to obtain performance measures, reliability factors, and other results.

The focus of this project was to design a system architecture that would include the fundamental underpinnings necessary for the practical application of such methods. At a minimum, such a system must support communication and interaction among submodels, include multiple formalisms and solution methods, and naturally allow parallel and distributed computation.

Existing Approaches

Existing commercial tools for supporting performance, reliability, and availability modeling are stretched to the limit by even relatively simple systems. Most available tools support only a single formalism and solution technique and do not allow the combination of submodels of different types. Support for parallel and distributed analysis is typically weak or non-existent.

Commercial firms have tended to invest solely in the user-interface aspects of modeling tools, rather than in improving the underlying technology. Conversely, academic tools typically

explore a few isolated, new techniques to the point necessary for demonstrating feasibility, but rarely develop a complete product—that is, a tool supported by adequate documentation, easy to operate user interfaces, on-line help, and training material.

Other than efficient combinatorial methods that apply only to restricted problems, the most dramatic advances in modeling capacity over the last few decades are based upon some form of stochastic decomposition. Thus one of our primary technical goals for the toolset is to provide several mechanisms for composing models from individual submodels, both for purposes of model specification and for applying efficient solution methods.

II. Technical Results

This section discusses some of the important technical results of the phase I effort. The first subsection describes the more important requirements that the eventual toolset must meet, and our initial approaches to satifying them. The toolset architecture is described in the next subsection, followed by a description of the constructed prototype.

Hierarchical models and availability of multiple formalisms

The ability to describe a complex system as a set of interacting models is a fundamental requirement for an integrated modeling environment. This goal can be accomplished in many ways. Certainly, at the user-interface level, we want to be able to break down a model into submodels in a hierarchical fashion. Each submodel can then be expressed in the most appropriate formalism. This solves the *specification* problem, but does not address the *solution* aspects. To address this deficiency, one of the Phase I proposal goals was to exploit the decomposition into submodels when solving, as well as specifying, the model.

As a result of developing and analyzing the current prototype, we have concluded that there are two fundamentally different ways of interaction between submodels, which achieve different goals.

When two submodels *A* and *B* interact through an *event stream*, they are conceptually connected in a cause-effect fashion. An event occurring in *A* might then cause a second event to occur in *B*, and vice versa. Hence, submodels *A* and *B* are simply a logical decomposition of a single, larger model. Multiple event streams may be needed between the two models, to represent different types of events.

In the example shown in Fig. 1, a queueing network model of traffic is being studied. A fundamental parameter of the top model, the number c_1 of servers in queue 1, is actually varying in time, since the servers in that queue are subject to failures. Instead of trying to include the failure and repair process in the queueing model, we can use a second model, this time a stochastic Petri net, to represent it. The two models can be developed independently, each of them at the desired level of detail. They are connected by the two event streams corresponding to failures (firing of transition *Fail*, arrival of a token in place *Down*) and repairs (firing of transition *Reinstate*, arrival of a token in place *Up*), respectively. These events in the Stochastic Petri Net (SPN) model correspond to events in the queueing network model, namely a reduction or an increase of the number of available servers in queue 1. The initial number k of tokens in place *Up* is the total number of servers.

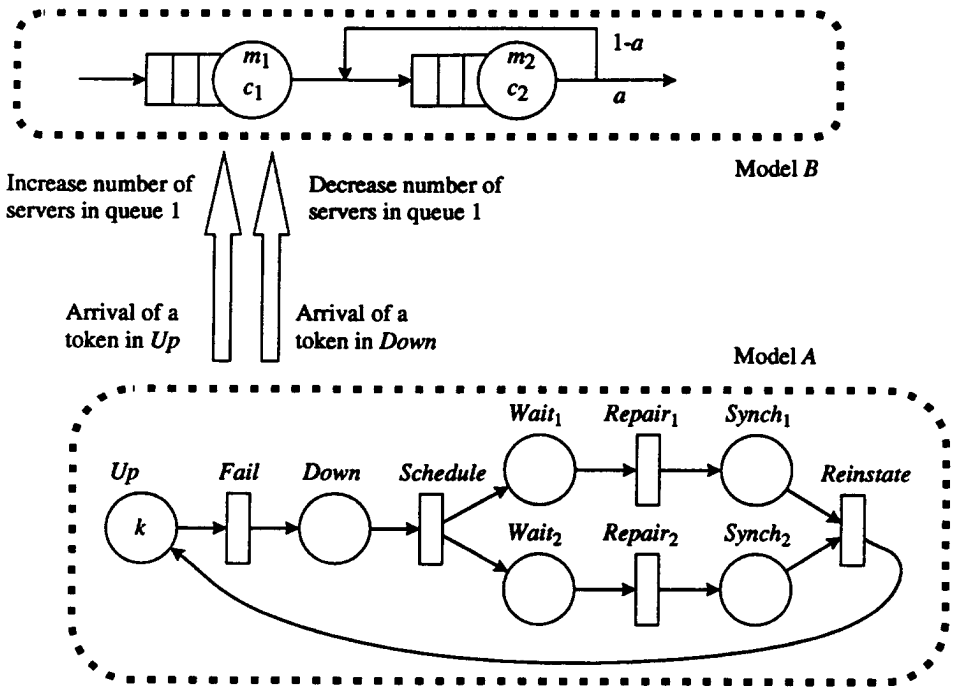

Figure 1: An example of interaction through event streams.

The solution of such a model can then be simply performed on the flat model, which is obtained by considering both submodels at the same time. The drawback of solving the flattened model is the size and complexity of the underlying state space and stochastic process, which grow rapidly as the number of submodels increases. If simulation is used, however, flattening the model is usually an effective decomposition approach.

The alternative method of interaction between submodels is *parameter exchange*. Using our example again, the two submodels *A* and *B* are solved independently. First, the failure and repair submodel is solved, to compute the following distribution:

P_1 = Prob(*i* servers are up in steady state) for each *i* up to c_1

or simply

E = E[number of servers up in steady state]

Then, this information is passed to the queueing network model, where it is used to approximate the number of servers in queue 1.

This latter approach can be described as batch, while the former is more interactive. The advantages of the batch approach are that it can be very naturally distributed, and each submodel solution is, in itself, much smaller. However, representing the interaction between submodels in such a static way–for example, using just a real number—can be more difficult on the user's part and often introduces an approximation.

A set of submodels interacting through event streams defines a single, stochastic model. The submodels can be defined using different high-level formalisms as long as the submodels can then interact by exchanging parameters in some standard way. For example, Fig. 2 shows a high-level decomposition of a model into four submodels–*a, b, c,* and *d*–which could, in turn, be composed of multiple submodels, etc.

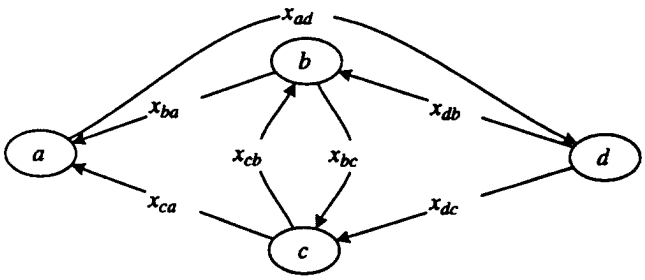

Figure 2: An example of interaction through parameter exchange.

The quantity x_{ij} represents the parameter, or set of parameters, computed from the solution of model i and passed to model j. Since there are circular dependencies, the solution of the overall model is obtained by starting with a guess for the value of some x_{ij} to bootstrap the iterations and then iterating until the values x_{ij} are stable within some given precision.

The following code shows how this iterative scheme is specified in the current prototype:

```
converge {              /* iterate until converged */
 real x_ad guess 1.0;   /* initial guesses */
 real x_bc guess 1.0;   /* initial guesses */
 real x_db := solve(d(x_ad),1);        /* compute first output of d */
 real x_dc := solve(d(x_ad),2);        /* compute second output of d */
 converge {     /* specifies an inner iteration */
  real x_cb := solve(c(x_dc,x_bc),1);
  real x_bc := solve(b(x_db,x_cb),1);
 }
 real x_ba := solve(b(x_db,x_cb),2);
 real x_ca := solve(c(x_dc,x_bc),2);
 real x_ad := solve(a(x_ba,x_ca),1);
}
```

Sharing of solution algorithms and efficient implementation

Due to the number of formalisms and solution methods necessary to support a variety of different kinds of analysis and problem domains, it is completely impractical to separately implement each solution method for each possible formalism. The toolset design must allow solution methods to be independent of the formalism used to define the model so that each solution method can be used on any appropriate model, regardless of the formalism used to express the model.

Our approach is to define a common intermediate representation, the Model Definition Language (MDL) and require each formalism to include a compiler for translating models into MDL. This allows us to apply a large array of algorithms for the solution of stochastic models to any high-level formalism, as shown in Figure 3. The only difficulty with this approach is ensuring that the translation to MDL does not lose important information which can be exploited to obtain an efficient solution. As a consequence, the most advanced solution algorithms can be incorporated into the proposed toolset and share their development cost among multiple formalisms. The final result is that the toolset will support a much larger class of systems, will

perform faster, and will use less memory than any other currently available commercial tools.

In the prototype, we began integration of the following solution methods:

- Steady-state analysis of a continuous-time Markov chain (CTMC)
- Transient analysis of a CTMC
- Steady-state analysis of a discrete-time Markov chain (DTMC)
- Transient analysis of a DTMC

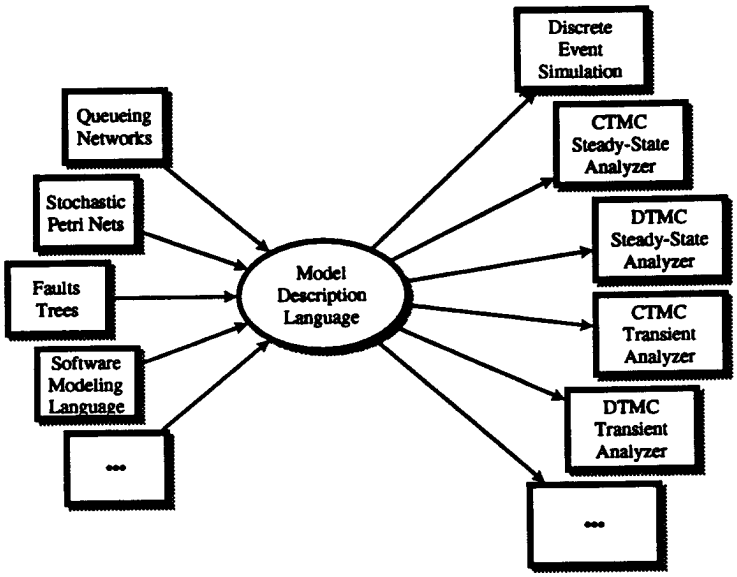

Figure 3: MDL as a method to integrate formalisms with solution methods.

Because algorithms exist to transform an SPN or a queueing network into a CTMC or a DTMC, provided it satisfies appropriate restrictions on the type of firing or service time distributions, the solvers listed above allow us to analyze such SPNs and queueing networks numerically. We also initiated work on a simulation engine which will also be applicable to these types of formalisms, independent of the type of distributions employed.

Using MDL allows us to define any number of formalisms on the front end and implement various solution algorithms on the back end, and the implementation effort will not grow quadratically as it would if we were working on individual tools instead of the integrated toolset proposed.

Representation and manipulation of random variables

A new requirement that evolved during the development of the prototype toolset was the importance of effectively representing and manipulating random variables.

We define the concept of *nature* as fundamental in the description of the quantities to be manipulated by the toolset. An object's nature can be:

- *Deterministic.* It has a set value.
- *Random.* It describes a random variable.
- *Process.* It describes a model (a stochastic process).

A deterministic value is exactly what we normally mean when we say value: 3, 5.2, TRUE.

Our models, however, have a fair amount of randomness in them, hence we need to be able to describe quantities whose value is only known by a probability distribution: these are random variables. A family of random variables indexed by a parameter (time) is then a stochastic process.

The current prototype allows several different operations to be performed upon random variables and stochastic processes. The tool checks for errors related to a variable's nature just as type checking is enforced in a programming language.

Language Specifications

One novel innovation of our approach is the introduction of a textual language for expressing and evaluating stochastic models. The language, SMART, is declarative in nature and contains constructs for defining and manipulating random variables and stochastic processes. The language syntax is in the style of the C programming language. An interactive SMART interpreter is one the primary user interfaces to our environment, but SMART programs can also be used to express complete models. SMART is described in detail in the appendix.

The Model Description Language (MDL) mentioned above is the other key language in our environment. It is used as a low-level model assembler language into which all or most high-level models can be mapped. MDL functions as the central, integrating component of the toolset architecture. The critical test of MDL is that it must be sufficiently general to represent models from a wide variety of formalisms, without losing the ability to exploit formalism specific knowledge to obtain an efficient solution. MDL is also described in the appendix.

Support for Model Composition

The mechanisms described above provide the structural underpinnings to support the integration of different submodels (including those developed with different formalisms) into a combined model (a model of models). Such techniques can increase the capacity to analyze complex models by one or more orders of magnitude.

The next logical step is to provide a means to define standard interfaces to submodels so that they can be effectively composed in variety of ways. The architecture provides a simple Model Specification Language (MSL) for defining interfaces to models. MSL describes only the interface, not the implementation of a model. The same language is used to define the external interface to all models, regardless of how they are constructed. In contrast, model implementations are described in a language designed expressively for a particular formalism, such as a reliability modeling language or a general queueing network language. MSL serves a role in our environment similar to that served by Module Interconnection Languages (MILs) in heterogeneous, distributed, software systems.

The key requirements related to model specifications are:

- Model specifications must contain all information needed to properly use the model.
- Models must be able to import and export parameters and events, with type checking support.
- The model specification should be strictly separated from its implementation.
- It should be possible for multiple models to conform to the same specification.

The specification formally captures the information necessary to use a submodel (its inputs and outputs, types, and constraints). This information can then be used automatically during model composition to construct the proper connections, check type conformance and constraints, and invoke the necessary toolset components (e.g., compilers, analyzers). The specification also defines available options for controlling the solution of the model, such as the degree of accuracy required from a numerical approach or the level of confidence required from a statistical approach.

Models can communicate in two very different fashions, by parameter exchange and by event exchange. Model specifications are the formal mechanism used to define the interfaces through which models exchange parameters and events.

The parameter exchange case is similar to that used to pass values to and from procedures in a programming language. Input parameters can be of various types, such as a scalar real value or even an empirical distribution represented by an array. Output measures are also defined with their names and types in the model specification, though their computation is deferred to the model implementation.

Parameters can also have a random variable or stochastic process nature. Such parameters can be used to implement an event exchange style of model composition. To fully support event exchange communication, a mechanism must be provided for defining possible events.

Specifications may contain constraints to ensure the validity of the model. Example constraints might restrict the range of values of a particular input value or ensure that an invariant condition remains satisfied. Constraints can be automatically tested during analysis to help ensure the integrity of the results.

Model specifications allow model composition to proceed at a relatively high level of abstraction. Maintaining a strict separation between model specification and implementation prevents higher level models from depending on the internal details of their submodels. This allows a submodel to be refined or drastically changed over time without impacting the development of the models that use it, though the final results or the computation resources required may be affected. In fact, the specification must not imply the formalism used to create the model. For this reason, the MSL contains no formalism-specific constructs. This separation is crucial to allow flexible model composition from individual submodels.

There are many reasons why it is desirable to have multiple models for the same specification. Often a model is refined over time as more information is known about the system or different aspects of system behavior are incorporated into the model. Because modeling always requires balancing competing cost and accuracy demands, it is frequently useful to maintain several different models for the same system: for example, a fast, low-cost, low-fidelity model for quick estimates and a more expensive, but higher fidelity model for more detailed analysis. If two such models conform to the same formal specification, it will be possible to easily switch between them as needed.

Another important reason why multiple models may exist for a single specification relates to the model refinement process. To allow traceability of results to the defining model, the toolset must allow version numbers to be attached to model implementations, computed results, and specifications. Thus, even if there is only one model for a particular specification, there will likely be several versions of that model over time.

Figure 4 illustrates the concept of model specifications. In this case, a single composite model uses two submodels, A and B, in its implementation. There are two choices for each submodel: two versions of Model A, and two different implementations of Model B. The

definition of the composite model does not depend upon which particular versions of each submodel are chosen; it will work with any legal combination. The composite model only requires that submodels meet the published specifications. Of course, different numerical results may be obtained with different combinations of submodels. The goal is to facilitate the process of obtaining those different results.

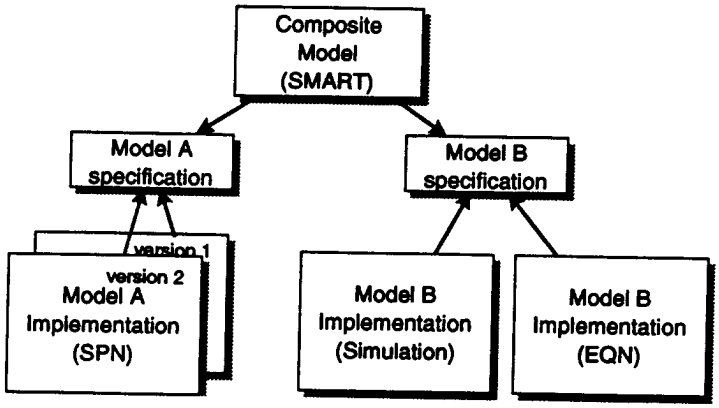

Figure 4: An illustration of model specification.

Extension to New Formalisms

The toolset will provide a set of standard formalisms, including stochastic Petri nets, queueing networks, fault trees, and various low-level, state-space-based formalisms such as CTMCs. These formalisms are general in the sense that they are not targeted to a particular application domain. A skilled modeling expert can use these general formalisms to represent systems from a wide variety of problem areas, such as computer systems design, communications, and flexible manufacturing system design. The most comprehensive formalisms, such as stochastic Petri nets, can be used, in principle, to model any discrete-state system. In practice, however, these formalisms alone will not suffice because they require a casual user to provide an excessive amount of detailed input, often in an unfamiliar context.

The solution is to provide custom formalisms to potential users, which will allow them to express their systems in the most appropriate and familiar language (possibly graphical). For example, a system engineer designing communication networks might want a language with primitives such as "bidirectional channel," "external load," and a built-in set of standard protocols. A designer of flexible manufacturing systems will want to express a model in terms of "cells," "part schedules," and "storage facilities." A reliability engineer will want to talk about "components," "failure rates," "repair facilities," and so on.

Clearly, however, it is not feasible or advisable to write a new tool for each class of customer. Instead the toolset infrastructure is designed to facilitate the rapid addition of new formalisms. The MDL concept aids tremendously in this regard. Since the speed with which add new formalisms can be added to the toolset will be a major factor in determining which markets to target, other methods will be developed to automate much of the process of adding new formalisms to the environment.

One promising approach is to define a Formalism Description Language (FDL), which is used to precisely define each formalism in the toolset, including how to translate its constructs into MDL. Once the elements of the formalism are described (including how to visualize a model expressed in the formalism, assuming that graphical editing capabilities are appropriate), a set of

software tools may be applied to translate this formalism description into a new instantiation of the toolset, with the new formalism as the main (or only, if so desired) user-visible interface. This approach builds on the successful compiler construction techniques that are now standard practice in that field.

One serious challenge to the automated addition of new formalisms is the need for the final translator to translate models into efficient MDL. This task is simplified somewhat by moving some general purpose optimizations into the general MDL analyzers. Early results indicate that this is feasible for state-space-based formalisms, which are to be solved using numerical methods based on the enumeration of the state-space or discrete-event simulation. More specialized formalisms might have specialized solution algorithms as well, and their efficient solution might indeed require a certain amount of custom implementation.

The proposed architecture facilitates the addition of new formalisms, even those requiring custom implementation that might not be defined efficiently using FDL. Once a compiler has been developed which translates models from the new formalism into the common MDL, models developed using the new formalism can immediately take advantage of all existing MDL analyzers. Even if such a compiler is developed by traditional techniques, it can reuse common expression parsing and other software shared by the other formalism compilers. Similarly, the graphical editors for new formalisms can share substantial software components with existing graphical editors. Finally, the object-oriented framework for the toolset facilitates customization by providing abstract, superclass standard components that new subclasses can extend.

Parallel computation of independent parametric models

Most modeling studies involve solving the same model for a large set of parameter choices. For example, we could solve a computer network system when the number of nodes in the network is 16, 32, 64, 128, or 256 (five choices), the speed of the communication links is 10 or 20 Mbit/sec (two choices), and there are three different network connectivity configurations. This results in 30 sets of parameters, each of them corresponding to a different system/model.

Since the 30 solutions are independent, an Experiment Manager is needed only to schedule each solution on an available workstation. If enough workstations are available, the speedup of this approach can be almost linear, where the solution time is the maximum among the individual solution times.

This simple approach to parallelism is not new. Existing systems implemented using the approach will be examined, analyzed, and, if possible, adapted for use in the proposed toolset.

Parallel simulation

Simulation is a statistical method; therefore it requires multiple runs to obtain accurate statistics (tight confidence intervals). The idea of parallel simulation is simply to execute N independent runs on N available workstations. This is similar to the approach described in the previous subsection, the only difference being that, instead of varying parameters, we are varying the initial seeds used by the discrete-event simulator to generate the streams of random events. Also, in this case, the speedup depends on the time for the worst-case run, but it can be considered almost linear in practice.

Again, this is a simple approach that has been considered before in the academic literature and has been implemented in academia, but no commercial tool has adopted it, despite its

simplicity.

Parallel solution of submodels in a decomposed model

As described above, decomposition of a model into submodels interacting through parameter exchange can drastically reduce execution time and memory requirements. Furthermore, the dependency graph among submodels immediately suggests a potential for parallelism. Consider, for example, the dependency graph in Figure 5. Submodel *a* must be solved first. Then, *b* and *c* can be solved next, in parallel. After *b* is solved, *d* and *e* can be solved, while *g* must wait for *c* and *d*. Finally, *h* can be solved.

Thus, this dependency graph naturally suggests a parallelism level of two (up to two submodels, *b* and *c*, and later *d* and *e*, can be solved in parallel).

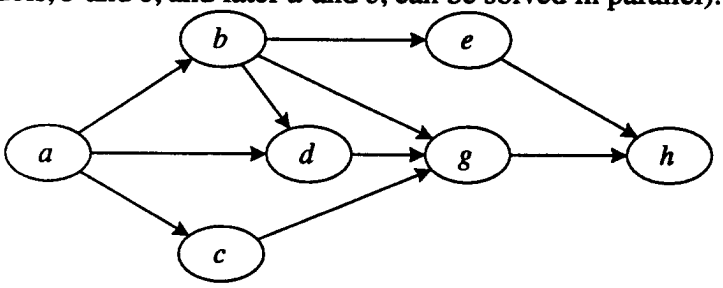

Figure 5: An example of inherent parallelism in a dependency graph.

Recognition of potential parallelism in the SMART language

The SMART language is a sequential, declarative language that is used to specify interactions among submodels and manage the required computations. However, independent declarations can be processed in parallel. For example, the statements:

```
real x := mtta(MyCTMC(lambda:= 1.0));
real y := mtta(MySPN);
```

compute `x` and `y` as the mean-time-to-absorption of two distinct models, one called `MyCTMC`,

presumably a CTMC with absorbing states, and the other called `MySPN`. Since the two models do not depend on each other, and `x` is not used in the call `mtta(MySPN)`, we can, in principle, recognize this parallelism and compute the value of `x` and `y` on different workstations. Recognizing this type of parallelism is feasible, although it requires care, since ignoring a dependency among declarations would result in erroneous computations.

A more challenging task is, perhaps, deciding whether the existing parallelism should be exploited. This is probably the case in our example, since the mean-time-to-absorption computation requires substantial execution time, especially if the size of the underlying processes is large. However, the syntactically similar statements:

```
real x := 3*lambda;
real y := expected(unif(1.0,2.0));
```

should not be parallelized because they require negligible computation. A heuristic for this decision must be developed, based on a guess of the required computation time and on the number of available workstations.

Distributed simulation

When the interaction among submodels is through event exchange rather than parameter exchange, the submodels are stochastically connected, and an exact solution requires, in general, to consider the submodels as a whole. Discrete-event simulation is a reasonable approach in this case, especially if the composite model is very large. However, the simulation time and the memory requirements might be excessively large, even for a single simulation run. Distributed simulation is a method of dealing with this problem, by assigning different portions of the model to different processors, or workstations, which then interact via messages (event exchange). The main challenge then becomes how to partition the model over the processors, especially since we want the partitioning to be transparent to the user.

Distributed analytical solution

Memory and execution time requirements are a challenge with analytical/numerical solutions just as much, if not more, as with simulation. Hence, the ability to use a distributed algorithm for the generation of the state space and the numerical solution of the underlying process for a model are very valuable. The results from a separate project at the College of William and Mary are very promising in this respect. They show that it is possible to employ several (5 to 10) workstations to work concurrently on the problem of generating the state space for a stochastic Petri net. The speedup is nearly linear, and, most importantly, the total amount of memory available overall among the workstations employed is available to store the state space of the underlying process. This reduces, or even prevents, the use of virtual memory, which is one of the main factors contributing to a slow sequential solution.

The challenges in applying this method are mainly in formalizing and implementing a general-purpose approach, which will be applicable to any formalism, and in achieving good speedups with larger number of processors (several dozens).

Extensible architecture

Another important requirement is that the environment be open and extensible environment as opposed to the closed monolithic architectures of many commercial modeling tools. The primary motivation for this emphasis on extensibility is to support new modeling formalisms and analysis techniques, as they become available, and to allow domain-specific interfaces to be developed that closely match the standard engineering tools used in different disciplines. An additional benefit of an extensible architecture is that it allows a phased approach for product development. For example, a core set of primary tools can be developed initially, followed by additional components as time permits. This approach is designed to reduce both risk and time-to-market.

Achieving these goals requires careful advance planning. In particular, strict attention must be devoted to the interfaces between components to ensure they are sufficient to support future components and do not contain implicit dependencies that would hamper future additions.

Current toolset architecture

The overall architecture is illustrated in Fig. 6. The modeler, using a workstation, interacts with three "SMART" managers. The SMART language has been adopted as the user-level conceptual interface because of its generality and flexibility.

The SMART editor manager allows to define input files containing multiple related

(sub)models, possibly expressed in different formalisms (in the figure we show stochastic Petri nets, queueing networks, and Markov chains, but this is not an exhaustive list). The SMART editor manager uses an ordinary textual editor for the description of the measures to be computed and for the definition of interactions between submodels, but it calls specialized graphical editors for the appropriate formalisms whenever possible, that is, when editing individual submodels. These graphical editors are all instantiations of GEGE, (generic environment for graph editing), a project being developed at William and Mary. GEGE, described in the appendix is not properly part of the toolset, since it only needs to be used to generate the editors used in the toolset, so it is not distributed as part of the toolset (it is shown used dotted lines in Fig. 6).

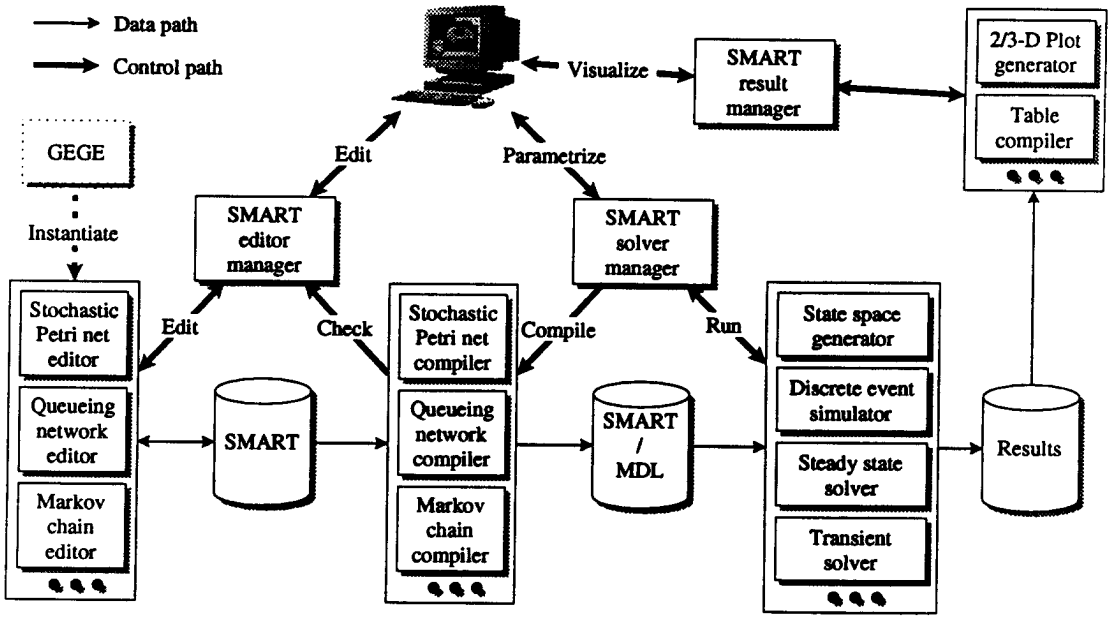

Figure 6: Overall architecture.

The SMART editor manager uses the various formalism compilers to check the syntactic correctness of the model being edited. These also ensure strict type checking, and enforce correct parameter usage for arrays and functions, thus catching many possible types of errors before model the analysis starts. Once a SMART model is saved and ready to be analyzed, the SMART solver manager can be invoked. It uses the appropriate formalism compilers to generate a low-level MDL model, still possibly containing multiple submodels. MDL is a unifying modeling language, general enough so that all high-level formalisms managed by SMART can be naturally and efficiently translated into it. The solver manager can then use any solver on this MDL model. In Fig. 7, we show a state space generator, a discrete-event simulator, and steady state and transient solvers, but, again, this list is not exhaustive. When the SMART solver manager has completed analysis of a model, the computed results are stored using a database.

These results can then be queried and displayed using the SMART result manager, which uses 2-D and 3-D plotting programs, and table generator facilities in the most common formats (ASCII, Postscript, latex, etc.).

The SMART solver manager is the most complex components of the toolset. One of its key features is its ability to manage the concurrent solution of multiple submodels, using a network of workstations. It requests the compilers to compile a SMART model, resulting in a

compiled MDL model. Then the SMART concurrency manager can use the sequential description of the model and extract its intrinsic parallelism. A "work unit" is the smallest set of computations that the SMART concurrency manager determines should be performed sequentially on a single processor. These work unit are piped to the SMART distribution manager, which dispatches them to the available workstations on the local area network where the SMART solver manager is running. Each work unit causes the computation of a partial result, which can be as simple as a single value, or a complex as a multidimensional set of values. The SMART distribution manager ensures that the partial results are computed, and stores them into a temporary local database, where all the values required for the solution of a model are kept during one solution run. The SMART solution manager selects from this local database the results which, according to the user requests specified in the SMART model, should be stored in permanent storage.

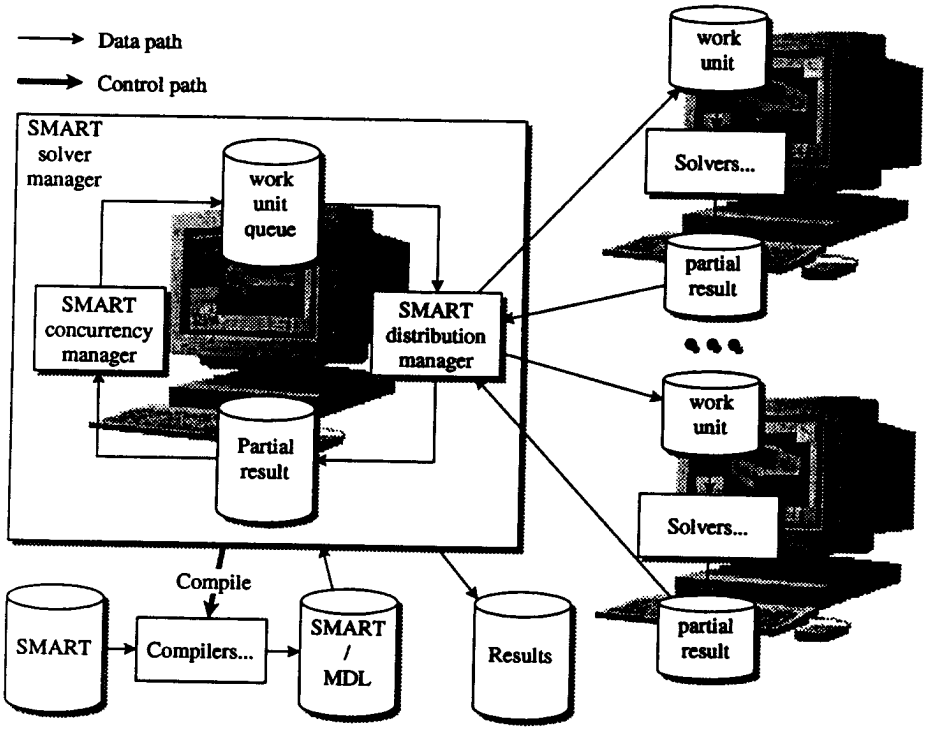

Figure 7: Solver architecture.

Examples of work units are:

- The computation of a single value.

The following SMART statement specifies that x is to be computed as the expected value of a random variable defined as the minimum of an exponentially distributed random variable with rate 0.2 and the constant 3:

```
real x := avg(min(expo(0.2),3));
```

Since x does not depend on any other value, it can be computed as an independent unit of work.

- The computation of an array of values.

The following SMART statement specifies to compute, in array f, the first ten factorial numbers.

```
for (int i in {0..9}) {
     int f[i] := if(i == 0, 1, i*f[i-1]);
```

Given the way they are specified, these ten values are computed sequentially, f[0], f[1], and so on. The SMART concurrency manager recognizes this and defines the computation of the entire array as a single work unit. Note that, had we used the statements:

```
int fact(int n) := if(n == 0, 1, n*fact(n-1));
for (int i in {0..9}) {
     int f[i] := fact(i);
```

instead, ten independent work units would have been generated. Most likely, this increase in parallelism would not be of any use, since the total amount of computation for f[9]alone would be equivalent to that required for the sequential computation of the entire array.

For a simple computation such as the factorial, the overhead of starting a remote computation outweighs the benefits of parallelism. Hence, in an actual implementation, a "work size threshold" should be achieved before a work unit is actually executed concurrently. The SMART concurrency manager must also label work units as "ready (to run)" or "waiting (for partial results)". A work unit u is ready if all the work units u_1, u_2, u_n which compute partial results needed to define the value of u have already been computed.

Consider for example the following SMART code:

```
real f(int iparm) := ...;
real g(real rparm, int iparm) := ...;
real h(real x) := ...;
real A := f(3);
real B := f(4);
real C := f(5);
real D := min(A,B,C);
real E := f(0);
real F := if(D < E, g(D,1), g(E,0));
real G := h(D);
```

The left portion of Fig.8 illustrates graphically the dependencies among the variables. An arc from a to b indicates that the value of a is needed before the computation of b can start. Nodes not on the same directed path may be computed concurrently. In the example, given enough available workstations, the value of A, B, C, and E can be computed concurrently at the beginning. Then, as soon as A, B, and C have been computed, D can be computed. When both D and E are computed, the computation of F can begin, while the computation of G can start as soon as D is computed.

In the current prototype, the correct behavior is achieved by scanning the model, generating the work units, as defined above, and connecting them in a dependency graph. A node without incoming arcs can begin execution immediately. A node a with incoming arcs from a_1, a_2, ..., a_n is placed in a waiting list, with a counter set to n; every time a node a_i with an arc to a is computed, the counter for a is decremented; when the counter reaches 0, a is ready to be computed.

This approach works correctly for acyclic dependency graphs. If a graph contains cycles, the nodes in the cycles are treated sequentially, the assumption being that the user has correctly

specified some kind of recursion. Consider for example the SMART code used to compute the first 100 Fibonacci numbers into the array V:

```
int FIB(int j);
for (int i in {1..100}) {
  int V[i] := FIB(i);
}
int FIB(int j) := if(j <= 2, 1, V[j-1] + V[j-2]);
```

Here every element of the array V, except the first two, depend on previous elements, hence no parallelization is performed. The entire sequence of statements constitutes a single work unit. Note that, in the right portion of Fig.8, the node corresponding to function FIB is shown in dotted lines. This is because the value of a function is never explicitly stored (it is not a partial result); however, since the function uses partial results (previous values of V in this case), it can be said that its computation must wait for the values of V.

Note that the same dependency graph would apply if we had the following (incorrect) definition for FIB:

```
int FIB(int j) := V[j];
```

Also in this case, the entire sequence of statements would constitute a single work unit. However, as soon as its computation would begin, the computation of V[1] would call FIB[1], which would in turn attempt to access the value of V[1].

Since this value is not yet known, a runtime error would occur. The detection of this type of problems is guaranteed by labeling each SMART value as "not-yet-computed" or "computed".

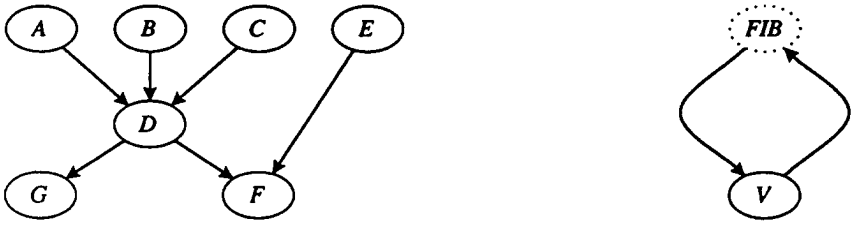

Figure 8 Dependencies among work units.

Prototype Toolset

The goals for the prototype activity were twofold: to test some of the basic feasibility assumptions behind the toolset and to develop an early working prototype for use in experimenting with various modeling scenarios. The first goal was achieved, and we conclude that multiple models can efficiently coexist and interact at various levels. Substantial progress toward the second goal was accomplished, but will continue as a separate project at William and Mary.

The prototype completed at the end of Phase I allows, through textual interaction, the specification of continuous and discrete time models resulting in a Markovian behavior. Multiple interacting models can be defined, and they can exchange parameters or events. A compiler for the textual language was developed, allowing us to test usage scenarios by describing complex systems from various domains.

III. Commercialization Results

The toolset and supporting products developed focus initially upon aerospace and defense

industries, where system failure can have devastating consequences, including loss of life. Specifically, the toolset supports the performance, reliability, and availability modeling needs of the aerospace and defense markets.

We intend to concentrate on reliability modeling initially. This market was chosen primarily because the current available modeling toolsets in these domains are lacking functionality in many ways, but the need for accurate timely quantitative analysis in high. We also have recognition and contacts in those communities.

Performance modeling is our second target, because it is a natural extension of the techniques for reliability, and because we have contacts and some recognition there as well. Outside the aerospace and defense sectors, the next most logical customer bases include computer and communication network designers. Secure funding for commercialization activities.

The ability of our toolset to support new formalisms easily will allow us to tailor the environment to the needs and notations of users in different application domains. Specific features, a user interface, usage scenarios, on-line help, and training material will support the notation, process, and output needs of aerospace and defense systems developers.

A commercial version of this prototype will require several accompanying support materials such as tutorials, libraries of component submodels, on-line help, graphical user interfaces, and user support. Development of such a toolset will require the resources typical of a complex software project: a product development plan, design documents, reviews, test plans, configuration management tools, and professional management and scheduling tools.

IV. Conclusion

There is an increasing need for powerful simulation and modeling tools to analyze the performance and reliability aspects of complex systems. This phase I STTR project resulted in the design and prototype implementation of an integrated extensible modeling environment which provides the functionality necessary to exploit dramatic improvements in analysis capability made possible by recent and classical advances in performance and reliability modeling. Such features include support for decomposition of complex models, communication among submodels, language constructs to represent and manipulate random variables and stochastic processes, and tools to manage parallel and distributed solution techniques. Two novel aspects of our architecture are the MDL and SMART languages described in the appendix. These languages provide much of the flexibility which makes this architecture unique.

Appendices to STTR Final Report

Gianfranco Ciardo
Department of Computer Science
College of William and Mary

October 18, 1995

Chapter 1

The SMART language

This chapter describes the SMART (Simulation and Markovian Analysis of Reliability and Timing) Language, which is the main textual interface for the user of the toolset. The SMART Language has the following goals:

- Performance, reliability, availability, and performability modeling are increasingly necessary steps in the design, development, and maintenance of complex systems, and we want to provide a unifying language for the definition and analysis of these models.
- Most, if not all, commercial software tools in this area are simulation-based, while other tools, mostly rooted in academia, use numerical solutions based on continuous time Markov chains (CTMCs) and requiring the solution of large systems of linear equations. SMART merges these two trends, allowing both simulation and numerical solution, plus some logical analysis as well.
- Recent developments have shown the feasibility of appying numerical methods to the solution of more complex stochastic processes [15, 16, 14, 11, 8, 7]. Hence, SMART greatly facilitates the description of these processes and can be used to experiment with various solution techniques for them.
- Since any model of a real system is likely to exceed the capacity of the machine (memory bounds) and the patience of the user (time bounds), fixed-point iterative techniques for the decomposition and solution of complex models can be easily specified in SMART. To the best of our knowledge, this is the first attempt to embed intop a tool such capabilities.
- A complementary approach when faced with large models is to use distributed algorithms. The SMART syntax is defined so that the inherent concurrency can be easily detected, and multiple processors, such as workstations on a network, can be used for the concurrent solution of a complex model.

Rather than in the particular syntax, the relevance of this presentation lies in the formalization of a mechanism to talk about multiple interacting stochastic processes (e.g., the number of processes waiting for the CPU), random variables (e.g., the number of processes waiting for the CPU at time 10), and deterministic quantities (e.g., the expected number of processes waiting for the CPU at time 10) in a formal way, and how to ask questions about them. Indeed, a SMART model usually contains many submodels in different formalisms, such as stochastic Petri nets (SPNs) or queueing networks (QNs). Strictly speaking, the actual syntax to express

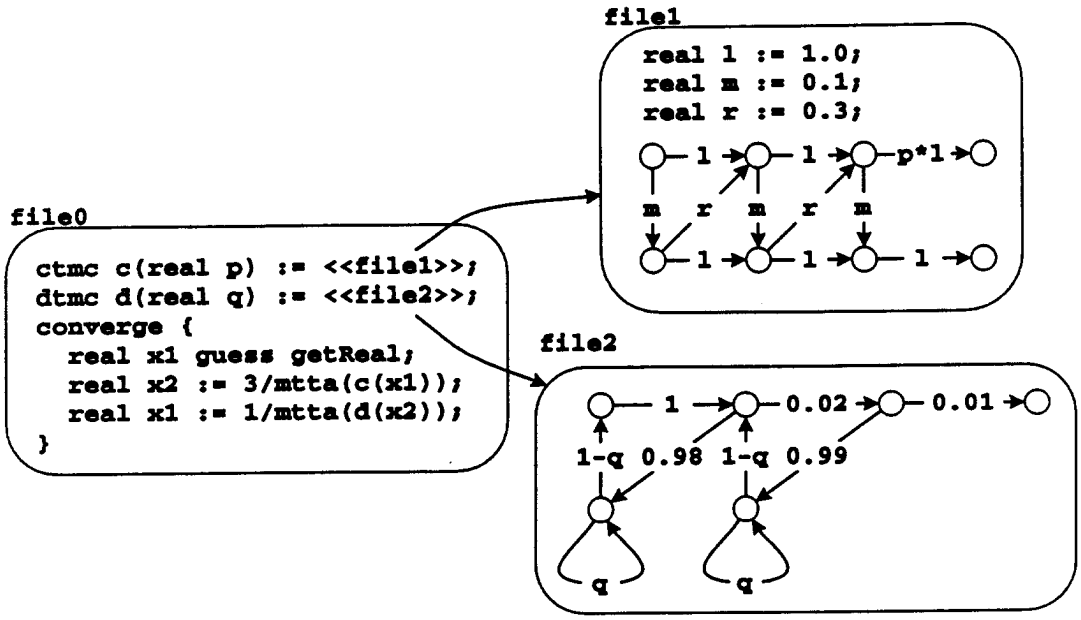

Figure 1.1: A SMART Language file with submodels.

a SPN or a QN is not part of the SMART Language, since each submodel can reside in a separate file which can be processed by an ad-hoc compiler. It is then best to see the SMART Language as the glue that allows a formal method of describing the interactions between the various submodels. Nevertheless, the syntax for each submodel will have a strong resemblance with the SMART language, especially in regard with the "expression syntax", since a user of the toolset will have to define expressions both within each individual submodel, and at the SMART level. This is illustrated in Fig. 0.1, where a file `file0` contains the top-level model information, namely, the definition of two submodels, and a fixed-point iteration over them. The exact structire of the submodels, a CTMC and a DTMC, respectively, is declared in separate files, `file1` and `file2`, respectively. The actual syntax of these two files is only required to have a "SMART flavor", see for example the definition of local parameters `l`, `m`, and `r` in `file1`.

SMART allows the direct description and solution of the following types of stochastic processes:

- Semi-Markov processes (SMPs) and their subclasses: independent SMPs (ISMPs), continuous time Markov chains (CTMCs), and discrete time Markov chains (DTMCs).
- SPNs and QNs. These high-level formalisms define an underlying generalized semi-Markov process (GSMP). In special cases, the GSMP might be a Markov-regenerative process (MRGP), a SMP, an ISMP, a CTMC, or a DTMC [11].

The most important characteristic of a model expressed in a given formalism is whether it describes a discrete-time stochastic process $\{X^{[k]} : k \in \mathbb{N}\}$ or a continuous-time stochastic process $\{X(t) : t \geq 0\}$. In SMART , two classes of discrete-state Markov processes are fundamental: DTMCs and CTMCs. In DTMCs, the state of the model is observed at each integer time, while, in CTMCs, events occur at any time, but the amount

of time elapsing between any two events is exponentially distributed. Such models can be combined in very natural ways, and are quite general, due to the concept of *phase-type* distributions, which is applicable both to the discrete and the continuous case. In both discrete-time and continuous-time models, the timing of events can have a mass at zero (immediate events) or at infinity (nonoccurring events). The underlying DTMCs and CTMCs can be solved numerically for steady-state or transient analysis. Alternatively, discrete-event simulation can be employed for the solution.

However, a model containing both discrete timing (other than zero or infinity) and continuous timing, defines an underlying process which can be independent semi-Markovian, semi-Markovian, Markov-regenerative, or generalized semi-Markovian. The numerical solution of these processes is sometimes possible depending, among other factors, on whether a transient or steady-state solution is desired. Discrete-event simulation is often the method of choice in these cases, especially for transient analysis.

SPNs and QNs are just two of the most commonly used high-level formalisms. Other high-level formalisms could be added. The only requirement for their integration is that they have an underlying stochastic process which can be managed by SMART.

1.1 SMART Language overview

SMART uses a strictly-typed declarative language. All types are predefined, that is, dynamic definition of new types is not possible, except for the creation of multi-dimensional arrays of predefined types.

A SMART Language file consists of a sequence of statements. Most of them declare or define (declare and specify the value of) a "function" of some set of "parameters". As a special case, the set of parameters can be empty, this is useful to declare constants. In this sense, "constant" simply means that the object being described is a function with no parameters, not to be confused with a non-random, or "deterministic", quantity, as opposed to a random variable or stochastic process. To ensure strict type-checking, the "type" of the function and of its parameters must be defined.

It is possible to define compound declarative statements, either to talk about arrays of values (the `for` statement), or to specify fixed-point iterations (the **converge** statement). The ability to specify arrays is particularly useful for parametrics studies, that is, when exploring how the results are affected by a change in the modeling assumptions, such as the initial number of tokens in a place of a SPN, or the value of a state-to-state transition probability in a DTMC. Compound statements can be nested.

Another type of statement is used to set SMART options. For example, there are options to control the behavior of the solution algorithms, or to interact with permanent storage. By default, the tool coomunicates results onto the "standard output". However, using the appropriate options, it is possible to write these values into an ordinary ASCII file, or into a custom-formatted file, for inspection by a specialized inspector utility. This utility can then be used to select, visualize, and further manipulate results stored in one or more files.

1.2 Deterministic SMART calculations

We start with several examples that use the SMART Language to perform simple "calculator-like" computations. This illustrates the power of the language without having to introduce random variables and the more complex

formalisms right away.

1.2.1 A constant: π

An example of a simple definition is

```
real pi := 3.14;
```

which defines the constant `pi` of type `real` with value `3.14`. Since the SMART Language is declarative, not procedural, the value of `pi` is now set for the rest of its life. A second definition

```
real pi := 3.1415;
```

causes an error. Indeed, even a second occurrence of exactly the same definition causes an error: each function can appear exactly once on the left-hand side of a definition statement.

1.2.2 A one-parameter function: the factorial

We can define a function to compute the factorial of an integer, $n! = 1 \cdot 2 \cdot \ldots \cdot n$, as

```
int fact(int n) := if(n==0,1,n*fact(n-1));
```

The predefined function `if` returns the value of the second or third parameter, according to whether the first parameter is `true` or `false`, respectively.

Note that we can call the function `fact` with either the Ada-like [29] named parameter notation, `fact(n:=7)`, or with positional parameter notation, `fact(7)`. Either mode can be used, as desired. The named parameter notation is especially useful to avoid errors and improve readability when a function has a large number of parameters.

1.2.3 Arrays: the first ten factorials

Since function `fact` has a parameter, its definition does not cause any computation to be performed. To request the value of the first ten factorial numbers, we can then use the following statement:

```
for (int i in {0..9}) {
  int f[i] := fact(n:=i);
}
```

The set of values specified for `i` is from `0` to `9`, with a step of one (the default). To refer to a particular value computed inside a `for` statement from a subsequent statement, the familiar "square bracket" array notation is used. For example, the value of `f` corresponding to `i` equal `7` is denoted by `f[7]`.

Since `for` statements can be nested, the dimensionality of a declaration is determined by the sequence of iterators, from the outermost to the innermost. For example,

```
for (real x in {0.1,0.2}) {
  for (int k in {1..5}, int i in {3}) {
    int g[x][k][i] := ... ;
  }
}
```

defines a tridimensional array g (of constants). The first iterator is a **real**. This is legal, but care must be taken to avoid indexing errors due to the finite precision of floating point arithmetic. The expression `g[0.1][1][3]`, for example, is well-defined. However, the expression `g[pow(10,-1)][1][3]` evaluates to `null`, an undefined value, unless `pow(10,-1)`, which is supposed to compute the power $10^{-1} = 0.1$, returns the exact same floating number representation as that used to store 0.1 on the particular machine on which the analysis is being run.

It is illegal to specify a function with parameters inside a **for** statement, as in the following:

```
for (int i in {0..9}) {
  int fact(int n) := if(n==0,1,n*fact(n-1));
  int f[i] := fact(i);
}
```

This is because such a function would be either independent of the iterators, as `fact` is in the above example, hence it could be specified outside the **for** statement, or dependent on them, in which case the semantic of calling it from outside the **for** statement would be ill-defined, since the iterators have no value outside the `for` statement. We could allow the definition of arrays of functions with parameters by allowing a statement of the form

```
 for (int i in {0..9}) {
  int f[i](int n) := ...;
}
```

but the same effect would be achieved by

```
int f(int i, int n) := ...;
```

at the top level, hence, we avoid this complication altogether.

In our example, we can actually get rid of the definition of the function **fact**:

```
for (int i in {0..9}) {
  int f[i] := if(i==0,1,i*f[i-1]);
}
```

This definition is the most efficient, but it uses the fact that the **for** statement is executed in order, from `i` equal 0 to `i` equal 9. If we had specified the set of values for `i` as `{9..0..-1}`, or in any other order, this definition would fail. Furthermore, it only defines how to compute the first ten factorial numbers, it actually computes them, but it would not help much if we needed the value 15!.

The scope of the iterators in a **for** statement is local to that statement and to those included in it. All other identifiers have global scope.

1.2.4 Parameter defaults

Normally, each formal parameter in the definition of a function must correspond to an actual parameter in a function call. However, it is possible to set default values for the formal parameters. If we define

```
int f2(int a:=1, int b) := a*b;
```

the following function calls `f2(a:=1,b:=3)`, `f2(1,3)`, `f2(a:=default,b:=3)`, `f2(default,3)`, or `f2(b:=3)` are exactly equivalent. Note that `f2(3)` is not allowed: named parameters must be used if some parameters are not listed explicitly. [1]

1.2.5 A parallelizable `for` statement

The `for` statement is a natural candidate for parallelization. Indeed, it is possible to recognize parallelism in SMART and perform the concurrent computation of all the values specified in a `for` statement, provided no dependencies from one "iteration" to the next exist,

For example, the 100 values

```
for (int i in {0..9}, int j in {0..9}) {
  int measure[i][j] := mymodel(i,j);
}
```

can be computed concurrently, This reduces the total execution time in cases where the solution of `mymodel` requires substantial computation, and multiple processors are available. Such situations are common in parametric modeling studies.

1.2.6 Overloading: a `real` factorial

The SMART Language allows the overloading of identifiers. The same identifier can be used to define more than one function, as long as the order, type, or name of the formal parameters can be used to distinguish which one is meant by a function call. For example, we could define a "real factorial", for `n` greater or equal one, as

```
real fact(real n) := if(n<1,0,if(n<2,n,n*fact(n-1)));
```

Then, `fact(8)` and `fact(5.4)` would call the standard (integer) factorial or the real factorial, respectively.

Overloading also applies to arrays. If we defined

```
for (int i in {0..9}) {
  int f[i] := fact(i);
}
for (real i in {1.0..9.0..1.0}) {
  real f[i] := fact(i);
}
```

[1] An alternative would be to use the same semantic as in C++, where any number of parameters at the end can be set to the default without having to list them explicitly in positional notation?

the expressions `f[2]` and `f[2.0]` would refer to the first and second array, respectively.

Examples of illegal overloading are:

```
int f(real y ):= ... ;
int f(real x) := ... ;
```

(it is impossible to decide which one is intended when positional parameters are used).

```
int f(real x, int i) := ... ;
int f(int i, real x):= ... ;
```

(it is impossible to decide which one is intended when named parameters are used).

```
int f(int i:=1) := ... ;
int f(real x:=1) := ... ;
```

(it is impossible to decide which one is intended by the function call `f(default)`).

The type returned by a function is not used to resolve ambiguities due to overloading.

1.2.7 Type promotions: manual or automatic?

Assume now that we defined an integer constant `k`, and we now want to call the real factorial function with `k` as the actual parameter. One reason for wanting to do this is that, if `k` is large, the integer factorial function might result in an overflow, while the value real factorial might still fit in the floating point representation of the machine.

We can force an automatic conversion by summing a (real) zero to `k`, as in `fact(k+0.0)`. This method works, but it is inelegant. A much better way is to use an explicit type conversion, `fact((real)k)`.

For simplicity's sake, most promotions such as the `int` to `real` promotion for `int k` in the expression `k+0.0`, occur automatically. Table 0.4 lists the automatic type promotions which can take place when evaluating an expression.

Promotion of the actual parameters in a function call or of the expressions used to index an array, however, requires special attention, because of the possible ambiguities arising from overloading. Consider for example the definitions

```
real x(int i, real j) := ...;
real x(real i, int j) := ... ;
```

The function calls `x(1,1.0)` and `x(1.0,1)` clearly refer to the first and second definitions, respectively. The call `x(1,1)`, however, could refer to either, depending on whether the first or the second actual parameter is promoted to `real`. First, an attempt to find a function requiring no promotion at all is made, that is, a function for which the formal parameters or indices exactly match the actual parameters or indices of the call. Only if this fails, promotion is attempted. If only one choice for promoting the types of the parameters or indices exists, promotion occurs, accompanied by a warning. If multiple choices are possible, as in the previous example, an error is issued.

1.2.8 Using iterators to define iterator values

The value of an iterator can be used to define the value of further iterators:

```
for (real x in {1..9}, real y in {x/2}) {
  real v[x][y] := x*y;
}
```

This is, of course, analogous to

```
for (real x in {1..9}) {
  real y[x] := x/2;
  real v[x] := x*y[x];
}
```

or to

```
for (real x in {1..9}) {
  real v[x] := x*x/2;
}
```

However, these simplifications would not be possible if multiple values were defined for y:

```
for (real x in {1..9}, real y in {x/2,0.1}) {
  real v[x][y] := x*y;
}
```

Syntactically, the iterators of a `for` statement must always be scalars: y depends on x in the above examples, but its dependency is implicit, that is, it would be an error to specify

```
for (real x in {1..9}, real y[x] in {x/2,0.1}) {
  real v[x][y[x]] := x*y[x];
}
```

1.2.9 Assertions: the null value.

Our real factorial function returns zero if it is called with an argument less than one. The integer factorial defined in Section 0.3.2 has an even worse behavior: it is caught into an infinite recursion if called with a negative parameter.

To prevent these problems, we can use the special value `null`, and modify our factorials as follows:

```
int fact(int n) := if(n<0,null,if(n==0,1,n*fact(n-1)));
real fact(real n) := if(n<1,null,if(n<2,n,n*fact(n-1)));
```

Attempting to perform any operation when one of the operands evaluates to `null` causes an error. Hence, the only way an expression evaluating to `null` does not cause an error is if it becomes an argument in a function call, and the function knows how to deal with this value. The predefined function `if`, for example, simply returns the value of the second or third parameter; if the selected one is `null`, so is the returned value. The same holds for the predefined function `case`. The predefined function `isNull` is also available. It returns `true` or `false` according to whether its argument has value `null` or not.

1.2.10 Fixed-point iterations: solving $x = x^{1/2}$

An important feature of the SMART Language is its ability to describe fixed-point type iterative solutions, where multiple parametric models are solved repeatedly, starting with initial guesses for their parameters, and successively refining them, until convergence within a given precision is achieved, or a maximum number of iterations is reached. For example, we can find the zero $x = 1$ for the expression $x - x^{1/2}$ by transforming it into the fixed-point equation $x = x^{1/2}$, and starting with any positive guess for x:

```
converge {
  real x guess 1000;
  real x := sqrt(x);
}
```

The value of `x` after the **converge** statement is the computed fixed-point value. The precision ϵ to which this value is computed can be changed in an option statement. The iterations are stopped when two subsequent values for `x` differ by less than ϵ (this is not equivalent to saying, and does not guarantee that, `x` is within ϵ of the true fixed-point solution).

We stress that, unlike the **for** statement, the **converge** statement does not affect the dimensionality of the objects declared in it.

1.2.11 Nested fixed-point iterations

It is possible, and often advisable, to nest fixed-point iterations. Consider the fixed-point iteration

```
converge {
  real d guess 0.5;
  real c guess 0.5;
  real b := fb(d,c);
  real c := fc(d,b);
  real a := fa(b,c);
  real d := fd(a);
}
```

The dependency graph of Fig. 0.2 represents this situation. An arc from x to y signifies that the value of x depends on the value of y.

However, we might know that `b` and `c` require many iterations to stabilize, but that, fortunately, each call to `fb` and `fc` requires little computation. Then, the following nested scheme might be advantageous:

```
converge {
  real d guess 0.5;
  real b guess 0.5;
  converge {
    real c := fc(d,b);
    real b := fb(d,c);
```

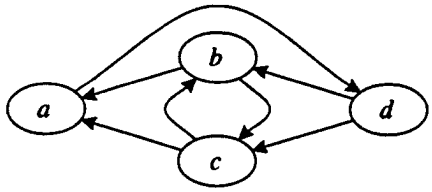

Figure 1.2: A nested dependency graph.

```
  }
  real a := fa(b,c);
  real d := fd(a);
}
```

now, the values of b and c are iterated upon until they stabilize, for each updated guess for d.

It must be possible to determine whether enough guesses are provided to start the iterations. The rule is simple: the dependency graph must be a directed acyclic graph (DAG) if all the nodes for which a guess is provided are removed. For example, the dependency graph of Fig. 0.2 must have guesses for one of the following sets of nodes: $\{a, b\}$, $\{a, c\}$, $\{b, c\}$, $\{b, d\}$, or $\{c, d\}$. It is legal to provide more guesses than needed (e.g., $\{a, b, c\}$), but only a subset of them are used.

Note that the position of the statement **real b guess 0.5;** causes the guess 0.5 to be used only once, the first time the innermost **converge** statement is executed. The subsequent fixed point iterations, every time the value of d is updated, will use the latest value for b, not 0.5. This second behavior would have been obtained by moving the statement **real b guess 0.5;** inside the innermost **converge** statement, but this would probably worsen the convergence behavior.

1.2.12 Nested for and converge statements

A for statement can contain a **converge** statement, as in

```
for (int i in {0..5}) {
  converge {
    real x[i] guess 1000;
    real x[i] := sqrt(x[i] + i);
  }
}
```

This specifies the computation of the six fixed point solutions to the six equations $x = (x+i)^{1/2}$, for $i \in \{0, 1 \ldots 5\}$. Consider now the statement

```
for (int i in {0..5}) {
  converge {
    real x[i] guess 1000;
```

```
    real x[i] := if(i==0,sqrt[x],sqrt(x[i-1]+i));
  }
}
```

This computes the fixed point solution `x[0]` as one, as expected. However, for any other value of `i`, no fixed point iteration is specified. For example, for `i` equal one, the "fixed-point iteration" is

```
real x[1] := sqrt(x[0] + 1);
```

This simply sets `x[1]` to $2^{1/2}$, since the value of `x[0]` is now set to one (possibly with a small error, since this value is computed numerically with an iterative procedure). However, the use of previously computed values is not always meaningless in a **converge** statement. For example,

```
for (int i in {0..5}) {
  converge {
    real x[i] guess if(i==0,1000,x[i-1]);
    real x[i] := sqrt(x[i-1] + i);
  }
}
```

uses the converged value `x[i-1]` as initial guess for `x[i]`. This is reasonable, since it could be expected that the value of the fixed point changes incrementally as `i` goes from zero to five. Only for the first one, `x[0]`, we really need to provide a guess.

While `converge` statements can appear inside `for` statements, the converse is illegal. [2]

1.2.13 Declarations and recursion: convolution

The difference between declaration and definition has already been discussed. Any number of declarations such as

```
int func(int n, real x);
```

can appear, but the definition of the same function must be eventually appear as well:

```
int func(int n, real x) := ... ;
```

The number, type, name, and order of the parameters in all the declarations and in the definition must match (otherwise they are assumed to refer to different, overloaded, functions).

Declarations accomplish the same goal as the "forward" keyword in Pascal or the templates of ANSI C. For large models, it might be desirable to place all the declarations at the top of the input file, postponing the definition of their values to a later section.

Declarations are needed when two function refer to each other recursively. For example, consider the computation of the normalization constant in a single-class closed product-form queueing network with load-dependent

[2] This might change in the future, since we might want to iterate on *arrays* of values.

servers. Using the notation of [28], assume that there are c_i servers with service rate μ_i at node $i \in \{0, 1, \ldots, m\}$, with relative visit ratio v_i. Then define $\rho_i = v_i/\mu_i$,

$$\beta_i(k) = \begin{cases} k! & \text{if } k < c_i \\ c_i! c_i^{k-c_i} & \text{if } k \geq c_i \end{cases},$$

and

$$r_i(k) = \begin{cases} \rho_i^k/\beta_i(k) & \text{if } k > 0 \\ 1 & \text{if } k = 0 \end{cases}.$$

If the total customer population is n, the joint probability of having k_i customers at node $i \in \{0, 1, \ldots, m\}$ is

$$p(k_0, k_1, \ldots, k_m) = \frac{1}{C(n)} \prod_{i=0}^{m} \frac{\rho_i^{k_i}}{\beta_i(k_i)}.$$

The computation of the normalization constant

$$C(n) = \sum_{k_0+k_1+\cdots+k_m=n} \left(\prod_{i=0}^{m} \frac{\rho_i^{k_i}}{\beta_i(k_i)} \right)$$

can be performed recursively defining

$$C_i(j) = \begin{cases} r_0(j) & \text{if } i = 0 \\ \sum_{k=0}^{j} C_{i-1}(j-k) r_i(k) & \text{if } i \neq 0 \end{cases},$$

so that $C(n) = C_m(n)$.

Let us now show how to compute the value of $C(n)$ using the SMART Language. The inputs m, n, c, v, and μ are read first, using the two functions **getInt** and **getReal**, which read an **int** or a **real** from the current input stream, respectively:

```
int m:= getInt;
int n:= getInt;
for (int i in {0..m}) {
  int  c[i]  := getInt;
  real v[i]  := getReal;
  real mu[i] := getReal;
}
```

Then, using the real factorial function of Section 0.3.6, we can define ρ, β, and r:

```
real rho(int i) := v[i]/mu[i];
real beta(int i, int k) := if(k<c[i],fact((real)k),fact((real)c[i])*pow(c[i],k-c[i]));
real r(int i, int k) := if(k==0,1,pow(rho(i),k)/beta(i,k));
```

Finally, we can define C with the following statements:

```
real C(int i,j);
real sum(int i, int j, int k) := if(k==0,C(i,j),sum(i,j,k-1)+C(i,j-k)*r(i,k));
real C(int i, int j) := if(i==0,r(0,j),sum(i-1,j,j));
```

Note that no computation is performed, since C is defined as a function, not an array. Normally, we want to save the last column of the tableau $C_i(j)$, hence we could define an array as follows:

```
for (int j in {0..n}) {
  real C[j] := C(m,j);
}
```

The above definition of C as a function is correct but extremely inefficient, because it causes the value $C(i,j)$ to be recomputed every time it is needed. The complexity is reduced to $O(n^2m)$ if the values $C(i,j)$ are stored in a table that allows direct lookup. Hence, we should change the definitions of `C` and `sum` as follows:

```
real sum(int i, int j, int k);
for (int i in {0..m}, j in {0..n}) {
  real C[i][j] := if(i==0,r(0,j),sum(i-1,j,j));
}
real sum(int i, int j, int k) := if(k==0,C[i][j],sum(i,j,k-1)+C[i][j-k]*r(i,k));
```

Now, the entries `C[i][j]` for `i` equal `m` contain the desired final column of the tableau, and there is no reason to save it as done before. To increase the efficiency, r, ρ, and β should be arrays as well. Also, we have ignored overflows and underflows in this example, although we did set the type of β to **real** instead of `int`, and used the real factorial, to postpone the occurrence of an overflow.

1.3 Random variables

The SMART Language can describe random variables with discrete or continuous time phase-type distributions, corresponding to the types **ph int** or **ph real**, respectively. For these, special operations are possible, and both numerical and simulation solution can be applied. Random variables with more general distributions are simply identified by as **rand int** or **rand real**, but they can be manipulated only in restricted ways, although simulation is always applicable.

1.3.1 Discrete phase-type distributions

The class $\mathcal{D}$ of (possibly defective) discrete phase-type (DDP) distributions can be informally defined as the distributions of the time to absorption in a generic DTMC.

More formally the following definition, from [9], is needed. A random variable X is said to have a DDP distribution, $X \sim \mathcal{D}$, iff there exists an absorbing DTMC $\{A^{[k]} : k \in \mathbb{N}\}$ with finite state space $\mathcal{A} \supseteq \{f,t\}$ and initial probability distribution given by $[\Pr\{A^{[0]} = i\}, i \in \mathcal{A}]$, such that states $\mathcal{A} \setminus \{f,t\}$ are transient and states $\{f,t\}$ are absorbing, and X is the time to reach state f (for "final"): $X = \min\{k \geq 0 : A^{[k]} = f\}$. If

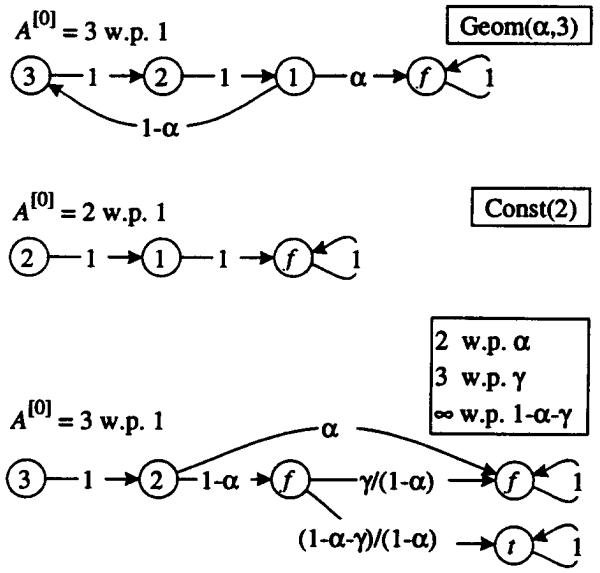

Figure 1.3: Examples of DDP distributions.

$\Pr\{A^{[0]} = f\} > 0$, the distribution has a mass at the origin, $\Pr\{X > 0\} > 0$. If $\Pr\{A^{[0]} = i\} > 0$ and state i can reach state t (for "trap"), the distribution is (strictly) defective, $\Pr\{X = \infty\} > 0$.

From this definition, it follows that $\mathcal{D}$ contains Const(0), Const(1), and Const(∞), and is closed under:

- Finite convolution. Given a finite set of independent random variables $\{X_i \sim \mathcal{D} : i \in \{1, \ldots n\}\}$, $X = \sum_{i=1}^{n} X_i \sim \mathcal{D}$.
- Finite probabilistic choice. Given a finite set of independent random variables $\{X_i \sim \mathcal{D} : i \in \{1, \ldots n\}\}$ and a pmf $\{\alpha_i : i \in \{1, \ldots n\}\}$, $X = X_i$ w.p. $\alpha_i \sim \mathcal{D}$.
- Finite order statistics. Given a finite set of independent random variables $\{X_i \sim \mathcal{D} : i \in \{1, \ldots n\}\}$, $X_{(i)} \sim \mathcal{D}$. $X_{(i)}$ indicates the i-th order statistic, that is, the i-th smallest value among $\{X_i : i \in \{1, \ldots n\}\}$ counting duplicate values. In particular, this includes $X_{(1)} = \min\{X_i : i \in \{1, \ldots n\}\}$ and $X_{(n)} = \max\{X_i : i \in \{1, \ldots n\}\}$.
- Infinite geometric sum. Given an infinite set of independent and identically distributed (iid) random variables $\{X_i \sim \mathcal{D} : i \in \mathbb{N}^+\}$ and an independent geometric random variable N, $X = \sum_{i=1}^{N} X_i \sim \mathcal{D}$.
- Multiplication by a natural number. Given a random variable $X \sim \mathcal{D}$ and $c \in \mathbb{N}$, $cX \sim \mathcal{D}$.

Fig. 0.3 offers a few examples of DDP distributions. If the distribution is not defective, the absorbing state t is unreachable given the initial probability distribution and it can be removed, as in the representation of Geom($\alpha, 3$) and Const(2). An example of a random variable with non-negative integer support which does not have a DDP distribution is N^2, where $N \sim \text{Geom}(\alpha)$.

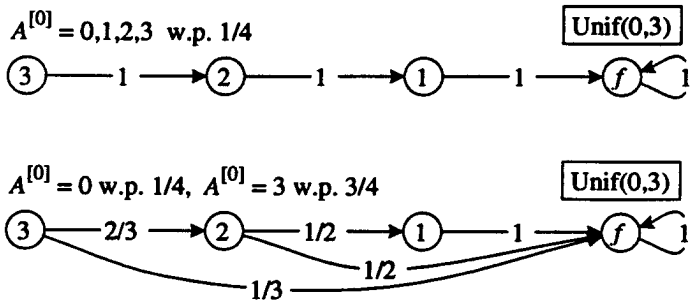

Figure 1.4: Equivalent representations for a DDP distribution

[9] shows that the "minimal" representation for a given DDP distribution is not unique. For example, Fig. 0.4 shows how to represent the distribution Unif(0, 3) in two different ways. Note that the "complexity" of the two representations is the same, if it is measured as a pair (n, a) where n is the number of nodes, four in both, and a is the number of arcs plus the number of states with positive initial probability, eight for both.

In the SMART Language, a random variable with a DDP distribution is defined using a predefined function, or operators. Also, any `int` value can be promoted to a `ph int`. Assume that `X` and `Y` are independent random variables defined as:

```
ph int X := geom(0.7);
ph int Y := unif(1,5);
```

Then, we can define:

```
ph int sumXY    := X+Y;
ph int prodNX   := 4*X;
ph int chooseXY := choose(0.3,X,0.7,Y);
ph int minXY    := min(X,Y);
ph int maxXY    := max(X,Y);
ph int geomX    := geom(0.1,X);
ph int sumXX    := X+X;
ph int chooseXX := choose(0.3,X,0.7,X);
```

However, definitions such as:

```
ph int diffXY   := X-Y;
ph int sumRX    := 4.0+X;
ph int prodRX   := 4.0*X;
ph int prodXY   := X*Y;
```

are incorrect because DDP distributions are not closed under these operations. A correct definition for them would be

```
rand int  diffXY   := X-Y;
rand real sumRX    := 4.0+X;
rand real prodRX   := 4.0*X;
rand int  prodXY   := X*Y;
```

The definition of **sumXX** and **chooseXX** requires special attention. The set of *independent* random variables involved in an expression must be recognized. In the above cases, this set contains only X, but only through a symbolic manipulation the expressions can be recognized to be equivalent to:

```
ph int sumXX    := 2*X;
ph int chooseXX := X;
```

which are DDP distributions.

A note about independence: every time a random variable is defined using predefined functions such as Bernoulli or Expo, it is assumed to be independent of every other random variable defined so far. On the other hand, whenever a random variable is defined using previously defined random variables, such as sumXY above, it is dependent on them. Hence, X and Y are independent, but X and sumXY of Y and sumXY are not.

1.3.2 Continuos phase-type distributions

A class $\mathcal{C}$ of (possibly defective) continuous phase-type (DCP) distributions can be defined. Its properties are analogous to those of $\mathcal{D}$. A random variable X is said to have a DCP distribution, $X \sim \mathcal{C}$, iff there exists an absorbing CTMC $\{A(t) : t \geq 0\}$ with finite state space $\mathcal{A} \supseteq \{f, t\}$ and initial probability distribution given by $[\Pr\{A(0) = i\}, i \in \mathcal{A}]$, such that states $\mathcal{A} \setminus \{f, t\}$ are transient and states $\{f, t\}$ are absorbing, and X is the time to reach state f: $X = \min\{t \geq 0 : A(t) = f\}$. If $\Pr\{A(0) = f\} > 0$, the distribution has a mass at the origin. If $\Pr\{A(0) = i\} > 0$ and state i can reach state t, the distribution is (strictly) defective.

$\mathcal{C}$ contains Const(0), Expo(1), and Const(∞), and is closed under:

- Finite convolution. Given a finite set of independent random variables $\{X_i \sim \mathcal{C} : i \in \{1, \ldots n\}\}$, $X = \sum_{i=1}^{n} X_i \sim \mathcal{C}$.
- Finite probabilistic choice. Given a finite set of independent random variables $\{X_i \sim \mathcal{C} : i \in \{1, \ldots n\}\}$ and a pmf $\{\alpha_i : i \in \{1, \ldots n\}\}$ over $\{1, \ldots n\}$, $X = X_i$ w.p. $\alpha_i \sim \mathcal{C}$.
- Finite order statistics. Given a finite set of independent random variables $\{X_i \sim \mathcal{C} : i \in \{1, \ldots n\}\}$, $X^{(i)} \sim \mathcal{C}$.
- Infinite geometric sum. Given an infinite set of independent and identically distributed (iid) random variables $\{X_i \sim \mathcal{C} : i \in \mathbb{N}^+\}$ an independent geometric random variable N, $X = \sum_{i=1}^{N} X_i \sim \mathcal{C}$.
- Multiplication by a nonnegative real number. Given a random variable $X \sim \mathcal{D}$ and $c \geq 0$, $cX \sim \mathcal{C}$.

Just as for DDP distributions, the representation for a given DCP distribution is not unique. Indeed, if we interpret the arc labels in Fig. 0.4 as rates instead of probabilities, we obtain two representations for a distribution obtained as a mixture of Erlang$(i, 1)$, $i = 0, \ldots, 3$, where we define Erlang$(0, \lambda)$ to be Const(0).

Assume that X and Y are independent random variables defined as:

```
ph real X := expo(0.1);
ph real Y := expo(0.2);
```

Then, we can define:

```
ph real sumXY    := X+Y;
ph real prodRX   := 4.0*X;
ph real chooseXY := choose(0.3,X,0.7,Y);
ph real minXY    := min(X,Y);
ph real maxXY    := max(X,Y);
ph real geomX    := geom(0.1,X);
```

However, definitions such as:

```
ph real diffXY   := X-Y;
ph real sumRX    := 4.0+X;
ph real prodXY   := X*Y;
```

are illegal because DCP distributions are not closed under these operations. A correct definition for them would be

```
rand real diffXY   := X-Y;
rand real sumRX    := 4.0+X;
rand real prodXY   := X*Y;
```

We stress that `ph int` and `ph real` values can be freely intermixed in an expression, but the resulting random variable is neither a DDP nor a DCP distribution: it has a mixture distribution, `rand real`, which can be managed only in limited ways (discrete-event simulation).

1.4 State-space formalisms

We now illustrate the formalism-dependent portion of the SMART Language, starting with the simplest formalisms, the state-space ones.

The specification of a state-space model consists mainly of the set of possible states, and of the timing and probability rules by which the model goes from one state to the next [17, 5].

1.4.1 Discrete-time Markov chains

The following statement specifies the DTMC `d1` (Figure 0.5), with a parameter `x`.

```
dtmc d1(real x) := {
  state a,b,c;
  init(a:1.0);
  arcs(a:b:1.0,b:a:x,b:c:1-x);
  real m1 := prob(at(inState(c),100));
};
```

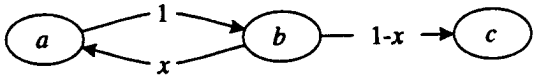

Figure 1.5: An example of a DTMC.

The definition of the "value" of the DTMC `d1` is done via a "block", that is, a sequence of statements enclosed in braces, used on the right-hand-side of an assignment. The statement `state a,b,c;` declares three "states". Since a state does not carry a value, the statement has the form of a declaration, but it is basically a definition.

The two subsequent statements are just expressions. Formally, this is legal because the two function calls return no value, we say that they have type `void`. The annotation operator ":" is used to annotate an expression, effectively creating a composite type. For example, `a:1.0` associates the real value `1.0` with the state `a`, while `a:b` indicates an arc from state `a` to state `b`.

Finally, the last statement, with a familiar syntax, defines a "measure", that is, a quantity which can be compute when needed. For example,

```
for (v in {0.1..0.9..0.1}) {
  real val := d1(v).m1;
}
```

computes, in `val`, the nine values corresponding to the probability of DTMC `d1` being in state `c` at time 100, when the formal parameter `x` varies from `0.1` to `0.9`.

The selector operator, ".", acts just like in ANSI C, where it is used to select a field in a structure. In our case, the "fields" are the identifiers declared within the block. For example, `a` denotes a state within the block, but the same state must be referred to as `d1(x:=...).a` outside the block.

Blocks can only appear at the top level and cannot contain within them a `for` or `converge` statement, or other blocks.

The block specifying a DTMC can contain at most one call to the following functions:

- `void init(state:real s1:r1, ...);`
  Sets the initial probability of state `s`i to `r`i. The call can be absent only if there is a single state or if the DTMC is ergodic and no transient analysis is requested. The probabilities must be nonnegative values, but they are not required to sum to one, since they are automatically normalized. However, at least one state must have positive probability.

- `void arcs(state:state:real s1:t1:r1, ...);`
  Sets the probability of going in one step from state `s`i to state `t`i in the DTMC to `r`i. If no arcs at all are specified from a state a, the state is assumed to be absorbing (an arc from a to a w.p. 1 is assumed). If the call is absent, all states are assumed to be absorbing. The probabilities of leaving a particular state a must be nonnegative values, but they are not required to sum to one, since they are automatically normalized.

- `void assert(bool b1, ...);`
  Defines a set of assertions, which must be true in each state. If, at any point in the execution, assertion `b`i does not hold, an error is issued.

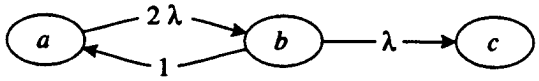

Figure 1.6: An example of a CTMC.

1.4.2 Continuous-time Markov chains

The specification of a CTMC is analogous to that of a DTMC. The following code defines the CTMC in Fig. 0.6:

```
ctmc c1(real lambda) := {
  state a,b,c;
  init(a:1);
  arcs(a:b:2*lambda,b:a:1.0,b:c:lambda);
  real m1 := prob(at(inState(c),100));
};
```

The compound statement defining a CTMC can contain at most one call to the functions `init`, `arcs`, and `assert` defined as for a DTMC. The only differences between CTMC and DTMC specifications are:

- The arc annotations represent rates, not probabilities, hence they are not normalized.
- Arcs from a state to itself are ignored, but a warning is issued.

1.4.3 Independent semi-Markov processes

The arc annotations in an ISMP are probabilities (possibly to be normalized), as in a DTMC, but the holding time in each a state can be arbitrarily distributed, instead of being deterministically equal one:

```
ismp i1 := {
  state a,b,c;
  init(a:1);
  arcs(a:b:1,b:a:0.9,b:c:0.1);
  holding(a:const(3),b:expo(20.0));
  real m1 := prob(at(inState(c),100));
};
```

The compound statement defining an ISMP can contain at most one call to the functions `init`, `arcs`, and `assert`, defined as for a DTMC, and to **holding**, defined as follows:

- `void holding(state:`x `s1:h1, ...);`
  where x is **ph int**, **ph real**, **rand int**, or **rand real**. Sets the holding time for state **s** to **h**. If a state has no outgoing arcs, no holding time needs to be specified, the state is considered absorbing with a holding time equal `const(infinity)`.

1.4.4 Semi-Markov processes

The most general state-space process that can be explicitly specified in the SMART Language is a SMP. In this case, a distribution is associated with each arc from i to j. There are two ways to define a SMP:

Race: given that the current state is i, specify the distribution $F_{i,j}(t)$ for the time required to go from i to j in isolation, that is, ignoring the other arcs leaving from i. The actual semantic is then that of a race among all the arcs leaving i, hence the holding time for i is implicitly assigned the distribution of the minimum of these times, over all the arcs leaving state i.

Preselection: choose according to a specified probability $p_{i,j}$ the next state j, given that the current state is i. Then, sample the distribution $F_{i,j}(t)$ to select the time required to go from i to j in isolation.

Figure 0.7 shows the equivalent ways to specify simpler processes (DTMCs, CTMCs, or ISMPs) as SMPs, with either method.

For example, the following specifies a preselection SMP **s1** exactly equivalent to the ISMP **i1** specified in the previous section:

```
smp s1 := {
  state a,b,c;
  init(a:1);
  arcs(a:b:1:const(3),b:a:0.9:expo(20.0),b:c:0.1:expo(20.0));
  real m1 := prob(at(inState(c),100));
};
```

Alternatively, we could have used a race specification for the arcs:

```
arcs(a:b:const(3),b:a:expo(18.0),b:c:expo(2.0))
```

The race specification is often more convenient and more natural. However, it does not specify how to "break the tie", and this might be a problem in the presence of discrete distributions.

The compound statement defining a SMP can contain at most one call to the functions **init** and **assert**, defined as for a DTMC, and **arcs**, an overloaded function:

- `void arcs(state:state:`x **s1:t1:v1, ...**`);`
  where x is **ph int**, **ph real**, **rand int**, or **rand real**. Sets the distribution for the time to go from state **s**i to state **t**i to **v**i, according to a race semantic.

- `void arcs(state:state:real:`x **s1:t1:p1:v1, ...**`);`
  where x is **ph int**, **ph real**, **rand int**, or **rand real**. Sets the probability and the distribution for the time to go from state **s**i to state **t**i to **p**i and **v**i, respectively, according to a preselection semantic.