Validating Personal Requirements by Assisted Symbolic Behavior Browsing

Robert J. Hall
AT&T Labs Research
180 Park Ave, Bldg 103
Florham Park, NJ 07932
bob-3GSTView04-@channels.research.att.com

Andrea Zisman
Dept. of Computing
City University
Northampton Square, London, EC1V 0HB, UK
A.Zisman@soi.city.ac.uk

Abstract

Risks and hazards abound for users of today's large scale distributed telecommunications and e-commerce systems. Service nodes are documented loosely and incompletely, omitting functional details that can violate stakeholder requirements and thwart high level goals. For example, it is not enough to know that a book finding service will locate a book for no more than a set price; will the chosen book vendor use an acceptable delivery mode and service? Will it retain or abuse personal information? The OpenModel paradigm provides the basis for a solution: instead of interface information alone, each node publishes a behavioral model of itself. However, large scale and multistakeholder systems rule out the use of traditional validation technologies, because state spaces are far too large and incompletely known to support concrete simulation, exhaustive search, or formal proof. Moreover, high level personal requirements like privacy, anonymity, and task success are impossible to formalize completely. This paper describes a new methodology, assisted symbolic behavior browsing, and an implemented tool, GSTVIEW*, that embodies it to help the user recognize potential violations of high level requirements. The paper also describes case studies of applying* GSTVIEW *in the domains of email and web services.*

1 Introduction

Existing and proposed approaches to validating the behavior of distributed systems are predicated on four shaky assumptions inherited from the validation of less complex systems:

A1. "Global" correctness properties exist;

A2. it is possible to formalize properties so that they can be automatically checked;

A3. models of all nodes are known completely; and

A4. the combined state space is "small enough", either to be searched exhaustively, or so that a "representative" set of behaviors can be checked.

For example, the SPIN model checker[12] exhaustively searches the synchronized product of node state spaces for violations of formal properties. Other approaches (such as [13, 3, 4, 7]) simulate a representative set of behaviors that "cover" the space in some sense.

Unfortunately, more and more distributed system applications fail to satisfy these assumptions[8]. Large scale multistakeholder networks, such as the email system and web service applications, consist of nodes controlled by distinct stakeholders often having conflicting goals. Goal conflicts imply stakeholders cannot agree on a common set of correctness properties (violating **A1**); instead, each stakeholder must judge for himself whether the system meets his own *personal requirements*. **A2** is violated particularly in e-commerce and telecommunications applications, where properties of interest to individual stakeholders, such as privacy and anonymity, are high level and impossible to formalize completely (see Section 2 for examples). Third, even though node stakeholders are willing to communicate some information about what their nodes do, it is not in their interest to reveal sensitive state information such as customer lists, cryptographic keys, or user preference profiles. Thus, **A3** is violated as well. Finally, **A4** is unrealistic for systems like email having millions of nodes, all of which can potentially interact with one another directly.

In this paper, we describe a novel approach to distributed system validation, called *Assisted Symbolic Behavior Browsing (ASBB)*, which can help stakeholders validate personal requirements within large scale multistakeholder distributed systems. It is implemented in a tool, GSTVIEW, which supports ASBB while avoiding dependence on assumptions **A1-A4**. The central idea of ASBB is to support the personal inspection of behavior classes that could arise from a particular task of interest to a stakeholder (avoiding **A1**). While many high level requirements are unformalizable, the tool supports *approximate formalization* in the form of automated checkers that draw attention to behaviors

Proceedings of the 19th International Conference on Automated Software Engineering (ASE'04)
1068-3062/04 $ 20.00 IEEE

that *may* violate requirements, reducing tedium and likelihood of error (and still avoiding **A2**). Behavior classes are represented symbolically, yet still with graphical visualization support, in order to tolerate the fact that some model data will be missing (avoiding **A3**). Finally, the symbolic simulation algorithm[5] deduces connections between relevant nodes dynamically, avoiding **A4** and the need to consider large irrelevant sections of the overall state space.

Because of the need to focus on personal requirements, we anticipate that ASBB based tools like GSTVIEW will be used occasionally by end users as well as designers and system analysts. This is because only the actual stakeholder can ultimately recognize all personal requirements. We believe that this type of end user validation will most likely be done either when a user first encounters a service or else when he wishes to execute a task having high risk or a high need for success. Routine, low-risk, or low-importance tasks will not need personal validation. These cases, as well as uncontroversial requirements sets, i.e. those shared by most or all users, can be validated (using GSTVIEW) by server administrators, designers or analysts, freeing end users from needing to do it for themselves. Thus, GSTVIEW is ultimately targeted at all stakeholders of a distributed system.

The key novel contributions of this paper follow. The structure of the paper mirrors this outline.

- We draw attention to the fact that validating large scale distributed systems raises new problems, including the unformalizability of high level requirements, the need to tolerate incomplete models, and the need to focus validation on the relatively small relevant portion of the behavior space.
- We describe the novel ASBB approach and the design of the GSTVIEW behavior browser tool.
- We evaluate these using two case studies, one an e-commerce service for book finding and the other a collection of email scenarios.

2 Two Case Studies

We introduce here two illustrative examples of multi-stakeholder distributed applications with their respective high-level requirements. Although not safety critical, these requirements can be "business critical" in e-commerce and telecommunications domains. The first, BOOKFIND, is an e-commerce web service application and the second, EMAIL, is a model of Internet electronic mail service.

BookFind Case Study. Figure 1 shows a representative instance of the BOOKFIND application. The **BookFinder** service attempts to locate, purchase, and deliver a book named by the user from one of a number of book vendors, each of whom may use one of a number of delivery services.

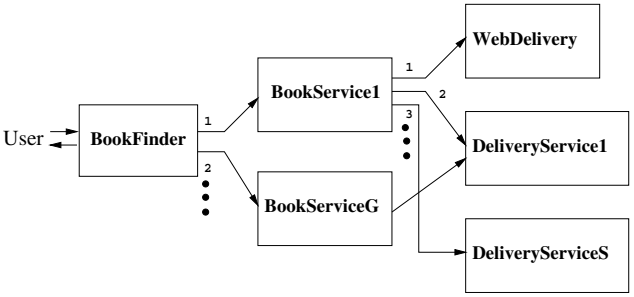


Figure 1. The BookFind Application.

The book vendors may have different book inventories, and the delivery services have different prices, capabilities, and time guarantees. As shown in the diagram, the top level bookfinder node queries its (dynamic) list of book vendors in order, purchasing from the first vendor found who has the book. In turn, a book vendor asks each of its (dynamic) list of delivery services in order whether it can meet the delivery constraints, accepting the first one that does.

The user states a task by giving book title, maximum book price, delivery time constraint (i.e. "urgent" or not), as well as email address, credit card number, and delivery address. **BookFinder** will either purchase the book or notify the user that it could not be obtained. The following three attributes of the input query have a bearing on how the order is filled: (a) whether the user's address is in the U.S.A. or outside it, (b) whether the user's address has a "p.o.box" number or not, and (c) whether the request is "URGENT".

The basic requirement is, of course, to get the book for at most the maximum price, but there can be other functional attributes of the service provided that can be important to the acceptability of **BookFinder** to a particular user: (BF1) whether a node (vendor or delivery service) will send spam email to the user after obtaining her email address; (BF2) whether the cost of delivery will be too much, which depends on just how important the book is to the individual; (BF3) whether a node will keep the credit card number after the transaction is completed, introducing a risk of identity theft; (BF4) whether a signature will be required to receive the book on delivery, which may be impossible for some users; and (BF5) whether the book will be delivered by "web delivery" instead of as a hard copy. (While rapid, this delivery mode does not result in a physical copy of the book, so may be unacceptable to some.) These are binary attributes of an individual user's preference, so it is possible to have at least 32 different preference sets for these attributes. Combined with the three input atttributes (a-c above), this yields 256 distinct personal requirements sets.

Another interesting type of undesirable behavior is "unexpected service failures" (BF6). That is, the user has the requirement that the book is eventually delivered, but everyone knows this can fail legitimately sometimes, for ex-

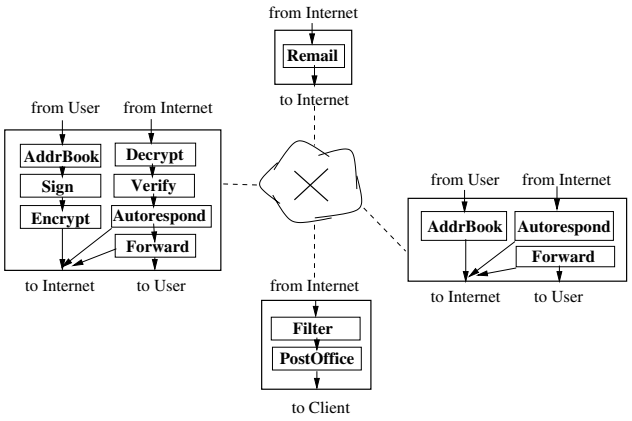


Figure 2. Email with 2 clients and 2 servers.

ample, when no vendor has the book. However, the service can also fail unexpectedly, such as when the chosen delivery service cannot deliver urgently to a "p.o.box" address. While it is straight-forward to construct a checker that can detect failure of the task, it is difficult to write one that can distinguish expected from unexpected failures.

Validation in this application illustrates the problems noted in the Introduction. Lack of agreement about global correctness (¬**A1**) is indicated by the diversity of opinion regarding BF1-BF5. For example, a vendor sending spam to customers is viewed by some as totally unacceptable, but by others as perfectly legitimate.

Furthermore, BF1 (spam) illustrates the practical impossibility of formalizing all high level requirements (¬**A2**): the spam filtering industry has been trying for over six years to automate the recognition of spam email from legitimate email and has failed. In BOOKFIND, there are legitimate non-spam email messages sent to the user, for example to send a notification of web delivery. However, pure advertising spam messages can be sent as well.

BOOKFIND also illustrates why model information will typically be incomplete during validation (¬**A3**): book vendors have an economic disincentive to publish in the model their entire inventory and price list, particularly for rare books. Such would enable competitors to undercut their prices or play other competitive games. They will, of course, answer single book queries, but this does not give competitors the same advantage, since it is impractical to query all books frequently enough to track changes. Other non-published information would include internal transaction numbers, sensitive customer information, etc.

Finally, since the **BookFinder** service can use any number of vendors, who can in turn use any number of delivery services, and since these lists can change dynamically, BOOKFIND illustrates the need to ignore large portions of the state space irrelevant to the current request (¬**A4**).

Email Case Study. Figure 2 shows a representative instance of the email application. Client hosts contain email processing features for a given user, like address book translation, encryption, or autoresponse. Server nodes can be standard post office style servers, which may provide spam filtering, or can be remailers which provide anonymity.

Each stakeholder can configure his node according to which features he desires active. For example, a user can turn on message forwarding or autoresponse ("vacation"), establish an account with the anonymous remailer, or set up encryption of messages. Administrators can set up user accounts as well as spam filtering based on addresses.

Moreover, for any particular message transaction, the preferences of the sender, server administrator, and recipient all matter to how the message is processed, so may impact the acceptability of the behavior to the message sender. Because of the parameterized nature of the features (e.g. forwarding is not just "on" or "off", but also specifies what address to forward to), there is an infinite space of possible personal configurations. There are a number of high level service quality requirements that one may have for email service as well: (E1) The message content must be delivered to the recipient; (E2) The message content must remain private everywhere except at the appropriate end points (sender and recipient); (E3) The recipient must not discover the sender's identity. In some cases, one doesn't care whether the message is private, while in others one does care. Similarly, one does not often demand anonymity, but when it is necessary, it is typically very important.

The EMAIL application also illustrates the validation problems mentioned in the Introduction. While many users expect their messages to remain private in transit, various governmental and other organizations expect to have access to message contents for various purposes; thus, there is no commonly agreed upon requirement relating to message privacy (¬**A1**).

E2 and E3 illustrate well the impossibility of completely formalizing high level requirements (¬**A2**). Any formalization of privacy must make unprovable assumptions about the difficulty of breaking the cryptography, who knows the keys, and actually what it means to "leak" information. If one sends a message through an anonymizer, the content of the message may expose the identity of the sender simply by containing some fact that could only be known by one person. Recognizing this case is impossible in general.

EMAIL also illustrates why model information will be incomplete (¬**A3**). No server administrator will expose the list of all valid email accounts, since this provides help to spammers. Similarly, no one will publish their own private cryptographic keys, since this would defeat their purpose.

Finally, EMAIL illustrates the scale issue (¬**A4**). Any user can send a message to any other user, so one should theoretically take a synchronized Cartesian product of the state spaces of all users' features, as well as those of the servers. But EMAIL has millions of nodes, resulting in an effectively infinite state space, with idiosyncratic structure.

3 ASBB and GSTView

As stated earlier, the central idea of *Assisted Symbolic Behavior Browsing (ASBB)* is to support the personal inspection for validity of behavior classes that could arise from a particular task of interest to a stakeholder. For each input task stimulus, a symbolic behavior tree, known as a *Generalized Scenario Tree (GST)*, is incrementally generated (here using the OMV tool[5]) that represents possible evolutions of the resulting distributed computation. The message passing and state changing inherent in this evolution is sequentially (along each tree branch as chosen by the user) visualized graphically for the user, and the symbolic expressions representing the detailed actions at each point are presented as well. The user must then examine these actions for desirability. Browsing is assisted by the use of automated checkers that can draw attention to behaviors that violate (or *may* violate) requirements, reducing tedium and likelihood of error. That is, formal descriptions of temporal properties are written that are then compiled into automated recognizers. Such formalizations may be exact or may only approximate the behavior set of interest. Visual indicators on-screen then indicate the status of each checker at a given point in the GST.

The rest of this section describes the design of GSTVIEW. First, we briefly overview the underlying modeling framework and symbolic simulation algorithm that drives the tool. Next, we describe the symbolic behavior browser's visualization, detail presentation, and tree overview functions. After that, we present GSTVIEW's support for approximate formalization. Finally, we discuss the methodology and process implied by GSTVIEW.

3.1 OPENMODEL and OMV

GSTVIEW is a tool for viewing GSTs. Thus, it requires a modeling framework and tool capable of incrementally building GSTs. Happily, the OPENMODEL project[16] provides such a framework and tool; see Figure 3. OPENMODEL is a modeling framework for heterogeneous multi-stakeholder distributed systems. In it, each node publishes an executable behavioral model of itself in a common interchange format, OPENMODEL Modeling Language (OMML)[6]. The OPENMODEL Validator (OMV) tool retrieves relevant models and integrates them to construct GSTs for a particular (symbolic) input stimulus. OMV is based on a *Symbolic Simulation* algorithm that builds trees by "pasting" together symbolic executions of individual node models. Models are published and retrieved via the OPENMODEL Distribution Infrastructure (OMDI). (Note that OMDI is still in the planning stages, but all other components of OPENMODEL have been implemented.) Further explanation of OPENMODEL and OMML can be found in references[6, 5].

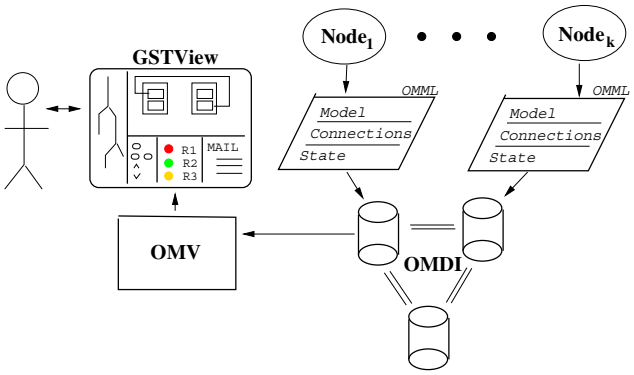


Figure 3. OPENMODEL **Overview**

OMML supports models written using arbitrary user-supplied function rich logics. Essentially, the model specifies within itself which logical theories it uses; each theory is represented within OMML as well and includes types, functions, and automated reasoning knowledge like axioms, simplification rules, and decision procedures.

OMV is implemented within the ISAT tool set[9], whose native specification language is P-EBF. P-EBF models are obtained by an automatic translation from OMML[6]. ISAT's automated reasoning tools support simulation (concrete and symbolic), term simplification, and semi-automatic proof. Models can be created in other validation tools and translated into OMML for publishing. For example, our BOOKFIND case study uses models originated in P-EBF and others originated in SCR's tabular notation. These are translated[6] to OMML and then translated out of OMML to P-EBF for use within ISAT.

Note that the ideas in this paper do not require OPENMODEL nor ISAT in a fundamental way. OPENMODEL could be replaced by another modeling framework, if desired. However, such other frameworks must handle missing information in the underlying models and dynamic integration of relevant models "on the fly". A GSTVIEW-like tool could be implemented in another environment, such as SCR or Statemate, assuming one has a complete distributed system model that is small enough to manage all at once. In such an alternative implementation, a mostly-unchanged GSTVIEW would interface with a new OMV implementation based in the native tool framework. Computed GSTs would be linear if indefinite branches are not tolerated by the underlying simulation algorithm. Of course, such environments could be extended by implementing their own symbolic simulation algorithms in order to compute branching GSTs.

3.2 Generalized Scenario Trees

A GST is a symbolic representation of a distributed computation. The root node represents the initial stimulus of in-

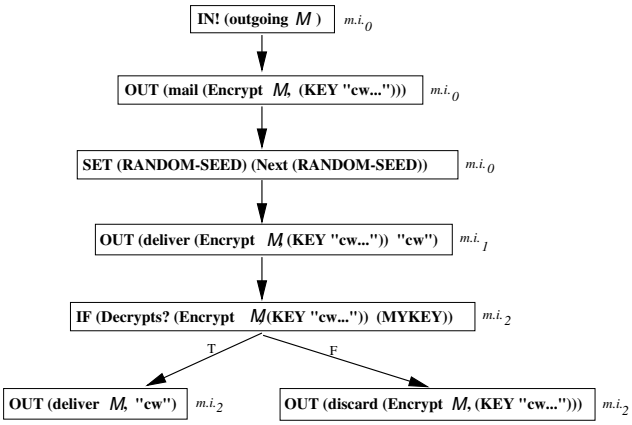


Figure 4. A simple GST in the email domain.

terest (for example, a book request or email message to be sent). This input event is then applied to a model representing the service node to which this stimulus is input. This results in a tree of possible responses, such as state changes (e.g. storing the user's request information for later use) and output events (e.g. sending a message to a book vendor to request the book). Note that each node may be indefinite (parameterized) if it was computed based upon placeholders for missing information. The tree may branch, if control within the node model depends upon the missing information. Figure 4 illustrates a simple EMAIL GST. Each node contains a symbolic expression with placeholders (e.g. `(MYKEY)`) for missing information. The `IF` node represents a control branch also caused by missing information. GSTs and their construction are described in [5].

3.3 Symbolic Behavior Browsing

GSTVIEW's symbolic behavior browsing is based on the capability to incrementally construct the tree of possible evolutions. Essentially, GSTVIEW provides an *overview window*, a *visualization window*, and a *detail window*, together with a *control panel* for guiding the evolution along lines of interest to the user.

The overview window (see Figure 5)[1] shows an abstract view of the currently computed nodes of the GST (not all will necessarily have been computed at a given time; they are computed on demand). As shown, the "current node" is the one in the small rectangle. Clicking the mouse on a node takes one to that node immediately, whereas the control window has other controls for moving within the GST.

As discussed in the next subsection, color is used to indicate the status of the automated approximate property checkers. (The colored dots to the right of a tree node

[1]Several of the illustrations in this paper use color, because color is integral to the design of the tool. The reader is advised to view this paper in color if possible; also, color screenshots are available on the supporting website available via the OPENMODEL project page[16].

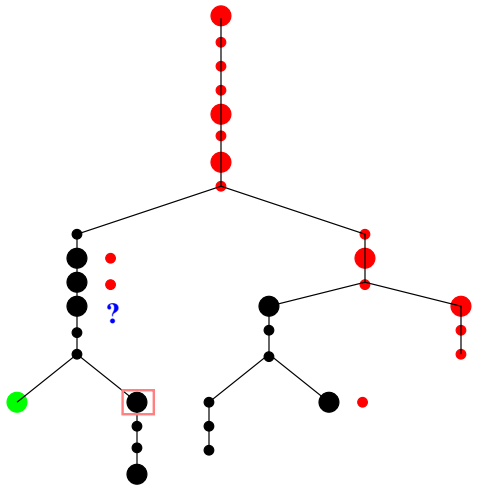


Figure 5. GSTVIEW**'s Overview Window**

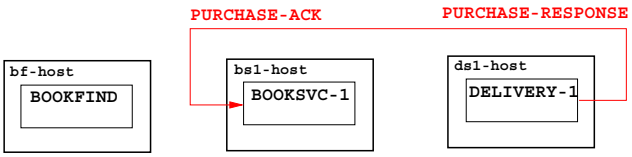


Figure 6. GSTVIEW**'s Visualization Window**

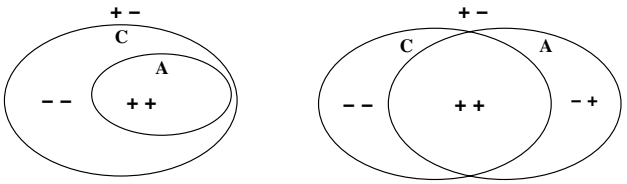
are also property indicators.) Large tree nodes indicate event nodes (message passing between model instances) and small tree nodes indicate other node types.

The visualization window (see Figure 6) shows the portion of the distributed system that is relevant down to the current node of the GST. It shows hosts and model instances that have processed at least one event along the path from the root to the current node. As one progresses down the tree, this display will be augmented with new model instances. If the current node represents an event passed between nodes, then a line is drawn connecting the two involved model instances and the line is labeled with the event types involved. (The declared connection information dictates how an output event of one node is translated to an input event of another.) Clicking on the items in this display brings up detailed descriptions of them. For example, clicking on the model name within a model instance brings up the specification of that node, while clicking on the event arc brings up a detailed description of the events and the connection between them.

Finally, the GSTVIEW detail window (see Figure 7) shows the symbolic expression representing in logical terms the occurrence (event, state change, conditional branch, etc) at the current node. This expression is produced by the symbolic simulation algorithm and simplified using term simplification and logical decision procedures. The user must inspect this to validate whether it represents desirable behavior. Note that in this case it is indefinite in that the con-

```
EVENT (PURCHASE-RESPONSE
        "ds1-host"
        "tx007"
        "SUCCESS"
        "Overnight -- SIGNATURE REQUIRED"
        (PLUS 1 (LOOKUP CONFIRMATION-NUMBER-CTR)))
```

Figure 7. GSTVIEW**'s Detail Window**



(a) Abstraction Approximation *(b) General Approximation*

Figure 8. Two approximation types.

firmation number is represented by a symbolic expression, since the delivery service node does not publish its current confirmation number counter value.

The size of GSTs is a potential concern, since users cannot be expected to inspect overly large or infinite trees. In our experience, for personal requirements relating to well-defined task classes, the resulting trees have been quite manageable. Equally important, however, is the fact that GSTView and OMV are designed to support *incremental* tree exploration. The user can choose which situations are of most interest and ignore others that are either unlikely to arise or low risk. Any ignored subtrees cost nothing, as they are never realized computationally. Finally, our automated checkers, as discussed in the next subsection, can radically reduce inspection time by focusing inspection on nodes that are more likely to be of interest.

3.4 Approximate Formalization

Browsing and inspecting each tree node for validity is time consuming and potentially error prone, given the human tendency to inattention and fatigue. Thus, it is well to provide automated assistance to the user by bringing to her attention situations that may violate personal requirements. GSTVIEW supports this through a general *requirement monitoring (ReqMon)* facility. Using this general facility, it is possible to express any computable predicate over tree paths. Consistent with our view that high level requirements are not accurately formalizable, ReqMons are not guaranteed to be sound or complete. Rather, the goal is to find good approximations, having neither too many false positives (indicated violations that are not) or false negatives (unindicated true violations). Figure 8 (a) shows standard abstraction-style concept formalization, where a complex (informal) concept C is approximated by a formal pure abstraction A. Anything recognized by A is a true positive (++), while anything falling outside of C is a true negative (+-). Instances within C but not within A are false negatives (--). False positives (-+) are not possible using this style. (An *exact* formalization would also lack false negatives, because then $C = A$.) Figure 8 (b) illustrates the more general approximation style we have studied in this work. Cases may lie within the scope of the formal concept A that are not valid C instances; these are false positives. For example, the predicate `exposes`, described below, is such an example. It attempts to formalize the notion of whether a text string exposes information from another text string. Our axiomatization has both false negatives and false positives, due to the impossibility of formalizing this concept exactly.

While our ReqMons are analogous to temporal logic expressions as are found in model checking tools, they are not the same. Our ReqMons are predicates over GST paths, the nodes of which consist of events and partial state changes. Temporal logic expressions are predicates on state sequences. Also, since we allow arbitrary functional logics and symbolic expressions, they are not propositional, being more akin to first order LTL than to PLTL. In particular, the notion of "next state" is not well defined here, since there is no explict global state of a single machine. However, we do allow computation of "eventually" and "globally" along a path, analogous to LTL's **F** and **G** operators.

The general ReqMon mechanism, whose details are suppressed here due to lack of space, is made more usable by providing macros that allow the declarative specification of different ReqMon types. These are declared in terms of logical predicates $P(\vec{v_n})$ that apply to GST nodes n. The logical expressions access the attributes of a GST node via special variables, denoted $\vec{v_n}$, corresponding to event operator, event parameters, state change value expression, etc.

G-ReqMons check properties that must hold true at every tree node. A G-ReqMon ("G" for global) is specified by a declarative predicate $P(\vec{v_n})$. A G-ReqMon raises an indicator if there exists a node on the path at which the reasoning system cannot prove P true. An example of a G-ReqMon used in EMAIL is

```
(or (not ?IS-EVENT-NODE?)
    (equal ?TARGETHOST "bobs-host")
    (equal ?TARGETHOST "andreas-host")
    (not (exposes ?EVENTARGS
                  (msg-content (msg...)))))
```

This expresses the fact for all event nodes, either the event must be directed toward the sender's host or the recipient's host or else it must not leak information from the message of interest. The logical predicate `exposes` is an approximate formalization of the concept of information leakage: it is true when a sizeable substring of the message content is detected among an event's arguments. (The names in all capitals refer to the specially bound variables $\vec{v_n}$.)

E-ReqMons check properties that must *eventually* hold at some point along each path in the GST. They too are specified by a predicate $P(\vec{v_n})$. They raise an indication if they reach the end of a path without having proven P true at a node on the path. For example, the following E-ReqMon predicate represents that the book must be delivered:

```
(and ?IS-EVENT-NODE?
     (equal ?EVENTOP "BOOK-RESPONSE")
     (equal "tx007" (nth 1 ?EVENTARGS))
     (equal "SUCCESS" (nth 2 ?EVENTARGS)))
```

("tx007" is the transaction id given as input to the request.)

N-ReqMons (Noticers), also specified by a predicate $P(\vec{v_n})$, raise an indication if they can prove P true at the node. This is different than a G-ReqMon for $\neg P$, because the reasoning system is incomplete: if P cannot be proven either true or false, an N-ReqMon will not indicate, whereas a G-ReqMon for $\neg P$ will. Noticers are useful in helping locate situations where a complex unformalizable requirement may be violated, but where it is too hard to formalize even an approximation to it. For example, in detecting when a book vendor may send spam, the Noticer

```
(and ?IS-EVENT-NODE?
  (equal ?EVENTOP "MAIL")
  (equal "bob@isp.net"
         (recipient (nth 0 ?EVENTARGS))))
```

indicates whenever any message is sent to the user's address. (The 0th argument to a MAIL event is the message.)

C-ReqMons (C for "Cleanup") are the most complex. They are defined in terms of two predicates $P(\vec{v_n})$ and $Q(\vec{v_n}, \vec{v_m})$ where n and m are GST nodes. Essentially, they are true if and only if for every node n_i on the path at which P was true, there exists a node m_i later on the path for which Q holds of n_i and m_i. A C-ReqMon was handy in the BOOKFIND case study in detecting when a vendor retains sensitive user information past the end of the transaction. For this example, the first predicate is

```
(and ?IS-STATE-CHANGE-NODE?
  (exposes ?VALUEEXPRESSION
           "1234-5678-8765-4321"))
```

where the 16 digit number is the credit card number of the user. The second predicate is

```
(and ?IS-STATE-CHANGE-NODE?
  (equal ?STATEVAR-1 ?STATEVAR-2)
  (not (exposes ?VALUEEXPRESSION-2
                "1234-5678-8765-4321")))
```

Here the `-1` and `-2` notation refers respectively to the first-matched node and the current node. This approximate formalization checks to see whether for any state variable that is set to a value containing the credit card number, it is eventually set to a value that does not expose that information.

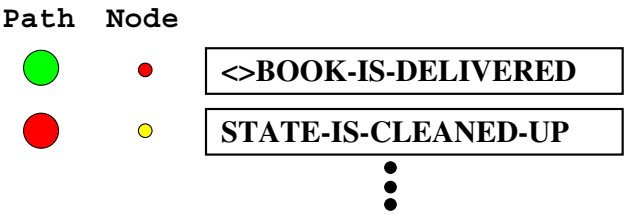


Figure 9. GSTVIEW**'s Requirements Window**

This is inexact both because the `exposes` predicate is inexact, and also because it can be fooled by incomplete reasoning over intervening state changes.

ReqMons are indicated in different ways on the GSTVIEW display. First, there is a small *requirements window* (see Figure 9) that gives indications for the current state of the various ReqMons. The "Path" column gives the status of the ReqMon on the current path. When the path is not complete, this will often be blank, since the path is not decided for E-ReqMons or C-ReqMons. For G-ReqMons, this is colored red (or yellow) if some node along the path has indicated a violation (or potential violation, in the case of incomplete reasoning). The "Node" column gives the status only of the ReqMon predicate at the current node. For example, for G-ReqMons, the state indicator is green, red, or yellow according to whether the predicate evaluates to TRUE, FALSE, or some other expression. Indications for other ReqMon types are similar. No indication is given unless some non-expected case has been detected. In the figure, the end of the path has been reached, so both the E- and C-ReqMons have been decided for the path, with the book having been delivered (so its indicator is green) but the state having not been cleaned up (so its indicator is red).

In addition to indication in the requirements window, the overview window's nodes are colored in summary fashion as well. Thus, a path with a red path-ReqMon will have the nodes of the path colored red. A completed path with no violated path-ReqMons will be colored green. Noticers are indicated by a blue question mark next to each node at which they indicated, while G-ReqMon violations are indicated with red or yellow dots next to the nodes found in violation or potential violation. See Figure 5.

Ultimately, the test of the usefulness of approximate formalizations is whether they are reasonably accurate in flagging situations that must be checked. In Section 4, we report on experience with the two case studies.

3.5 ASBB Process and GSTVIEW

Initial experience with the tool gives us initial ideas about a process for personal requirements validation using GSTVIEW. First, the current task of interest is specified in the form of an input stimulus (or sequence thereof), in as much detail as is known. In the BOOKFIND case study, this

would mean specifying the book title, maximum price, delivery time constraint, etc. Then, GSTVIEW is commanded to expand the tree a reasonable amount. For small trees (all those encountered here), this implies completely constructing the tree, but for large or infinite trees, this is bounded at some point (e.g., using a maximum depth cutoff). Then, the overview display is examined for indications of violations or potential violations of ReqMons. The user immediately inspects any nodes indicated by the tool. For Noticers and G-ReqMons, this means clicking immediately on each indicated node. For path ReqMons (E- and C- type), this implies inspecting the paths to understand the conditions bringing about the violation. (Typically, one inspects the branch points to determine what control predicate values led to the path.) Otherwise, one simply browses the tree in preorder traversal style, possibly ignoring subtrees that are not of interest (if they cannot arise in practice for some reason, say) and inspecting each node for violations.

Clearly, it is challenging to specify approximate (or exact) property checkers. We would not expect all end users to define these themselves. Relatively expert users or domain definers could create these over time, possibly in response to community experience communicated via discussion groups, and share them in libraries and domain ontologies. This amortizes the effort in creating them across potentially many uses. Once they are thus available, users could select them for use on a given task. Some parameterization may be required, which could be facilitated by the use of "wizards" that help people provide task- and requirement-specific information. These ideas are good candidate topics for future work.

4 Evaluation

While the GSTVIEW tool interface still has room for improvement, we believe a preliminary evaluation of the ASBB methodology and GSTVIEW is appropriate. In particular, we wish to evaluate our central hypotheses. First, symbolic behavior browsing is advantageous for validating high level requirements in complex systems where issues are hard to formalize and not known in advance to be relevant. Second, automated assistance in the form of approximate formalizations is useful in reducing the effort to find violations of some high level requirements.

Substantiation of Claim 1. Symbolic behavior browsing shows three main advantages for validation in our case studies. First, it allows one to find violations of unformalized requirements. Second, it allows one to find violations of requirements one has but did not know in advance were even relevant to the task. And third, it is able to prune the space of possible behaviors by using constraints derived from the particular task description relevant to the user at the moment (analogous to optimization by partial evaluation in the compiler world).

Scenario	Nodes	Violations	Time
0	65	BF1,BF3-BF5	0.7
1	63	BF1,BF3,BF5	0.7
2	65	BF1-BF5	2.2
3	63	BF1,BF3,BF5	2.1
4	76	BF1,BF3-BF5	2.2
5	94	BF1,BF3,BF5,BF6	3.9
6	76	BF1-BF5	2.7
7	94	BF1,BF3,BF5,BF6	3.6
Indefinite	144	BF1-BF6	5.6

Table 1. Statistics for BOOKFIND study.

In the BOOKFIND study, we used GSTVIEW to validate eight distinct scenarios (corresponding to the 2^3 combinations of input attributes discussed in Section 2). This resulted in discovering several violations each of the five personal requirements (BF1-BF5) from Section 2. A good example of unformalizable requirement for which we discovered violations is BF1. As we have discussed, since there is no (known) way to formalize whether a message is spam, it would have been impossible to use techniques like model checking to discover these violations; only through actually seeing the generated messages in context can the user judge this a problem. Here is the display in the GSTVIEW Detail Window for the GST node at which a violation occurs:

```
ACT MAIL
  (make-message
    (LOOKUP CUSTOMER-SERVICE-ADDRESS)
    "bob@isp.net"
    "Check out our self-help books!")
```

Distinguishing these nodes from those where transaction acknowledgement emails are sent requires the human's personal definition of spam.

Other violations came as "unexpected issues" discovered only through inspection of the possible behaviors. For example, there is no mention in the user interface description of the top level **BookFinder** node (see Section 2) of the possibility that the book may be delivered electronically, or of the possibility that a signature may be required to complete delivery. The user instead must discover these by browsing to the points in the tree where these issues are first encountered. All of BF1-BF6 are of this nature.

Finally, each of the eight GSTs was significantly smaller than the GST that is obtained from a completely unspecified input query. Table 1 summarizes the BOOKFIND experience. Each row shows for a particular scenario the number of GST nodes, which personal requirements violations were found, and the time (in seconds) to compute the tree. Note that the use of input constraints cut tree size by about half on average compared to the size of the tree resulting from a totally unconstrained input condition. The computation

Scenario	Nodes	Violations	Time
0	24	E1,E2	0.4
1	548	E1,E2	12.0
2	179	E1,E3	3.2
3	46	E1,E3	1.1
4	76	E1	0.4
Indefinite	8	—	0.1

Table 2. Statistics for EMAIL study.

times shown do not include inspection times.

In the EMAIL study, the usefulness of behavior browsing is even greater, because so much less information is present in the de facto service interface of email message transfer. Each potential recipient of a message may have different parameterized feature settings that could cause unforeseen problems. For example, just focusing on the features and requirements in this small study for one particular sender/server/recipient combination, a simple message traverses nine processing components, each of which is parameterized. Each different parameter setting, such as where to forward a message or what to say in the autoresponse message, can have subtle effects on requirements satisfaction, with a potential for many undesirable interactions[7]. Of course, none of the recipient's parameter settings are available in the email interface description, so the only way to discover some potential problems is by browsing behaviors that could arise in a particular task.

Examples of unformalizable requirements in EMAIL include E2 (privacy) and E3 (anonymity) of Section 2. Unexpected service failures (E1) can arise as well, such as when one's message is incorrectly treated as spam by a filter.

Table 2 shows the results of using GSTVIEW on five email service scenarios. Note that not all of E1-E3 were relevant for all of the scenarios. We have chosen five scenarios involving different combinations of personal sender requirements. All five require message content delivery. Two require message privacy en route. Two others require that the sender's true identity be kept from the recipient. In each case, we chose a "natural" level of information for each host in the system. That is, the sender knows everything about his/her own components, but only externally knowable information about other nodes. The "Indefinite" tree is of particular interest, because it is so small. Lacking address information in the input message means that OMV cannot deduce where to send the message, so GST computation is prematurely halted (and validation cannot succeed). Thus, by exploiting known details of the particular task, not only is the tree reduced in some cases (as in BOOKFIND), but in others the tree is enlarged enough to include violations. Clearly, validation approaches that consider *all possible* places to send the message are intractable.

In summary, both case studies provide evidence for the claim that symbolic behavior browsing enables finding violations of personal requirements that otherwise would be difficult or impossible to find using other approaches.

Substantiation of Claim 2. We wish to evaluate whether approximate formalization significantly helps speed up the search for violations. Clearly, if a requirement is not violated anywhere in the GST, then an automated checker cannot help speed up inspection. It may, in fact, hurt the time slightly when a false positive causes one to inspect a node that might otherwise not be inspected (if it lies on a branch that cannot arise in practice, for example). However, if a violation is present, then an approximate checker may help significantly, as long as false positives are not too numerous. False negatives, of course, neither increase nor decrease browsing time.

To measure browsing (inspection) effort, and therefore how much effort can be saved by speedups due to automated checkers, we first assume that the user inspects the tree in preorder fashion. (Any other uniform inspection order could be chosen.) We next assume that the effort to inspect one tree node is the same as any other. While this is in general not true, we believe it is close enough to provide insight. We then measure the effort a user would expend in inspecting the tree as simply the number of nodes inspected until the first node is reached that exhibits some sort of undesirable behavior. If no automated checkers are present, the effort is measured simply as the position in preorder of the first violation. Automated checkers, however, draw attention to nodes immediately, not requiring the user to traverse all the preceding nodes in preorder. Thus, the user is assumed to inspect the nodes indicated by automated checkers *first*, before any others. (And these indicated nodes are themselves inspected in preorder relative to each other.) Again, the user stops at the first true violation discovered. If no true violations are discovered, the other nodes (those not indicated by checkers) are then inspected after the indicated nodes.

In BOOKFIND, we specified Noticers for BF1 and BF2, a C-ReqMon for BF3, and an E-ReqMon for BF6. (BF4 and BF5 were not formalized.) In EMAIL, we specified G-ReqMons for message privacy (E2) and anonymity (E3), and an E-ReqMon for message delivery (E1). To measure how much inspection effort is saved by a ReqMon, we will count the nodes that must be inspected for each scenario in order to find the violation. We will treat the requirements independently (i.e. as if each is the only concern of the user). Table 3 shows quantitative results, aggregated across all eight scenarios, for BOOKFIND as well as aggregated results for EMAIL.

For each ReqMon, the first column gives the numbers of true positives (true violation detections), false positives (incorrect detections), and false negatives (undetected true violations). The second column expresses the rate of false positives among all tree nodes. The third column expresses

ReqMon	++/−+/−−	−+ Rate	Effort Savings
BF1	21/12/0	2%	91%
BF2	3/19/0	3%	95%
BF3	29/0/0	0%	96%
BF6	2/19/0	3%	98%
E1	18/0/0	0%	91%
E2	37/60/0	7%	99.6%
E3	7/3/1	0%	80%

Table 3. ReqMon statistics for both studies.

the effort savings as a percentage of total effort in the absence of the ReqMon (i.e. the cost of manual inspection up to the first violation). Note that the percentage is the savings due to the ReqMon, only considering trees where a true violation is present. This shows that clearly when a violation is present in the GST, it is of great value to have even an approximate ReqMon to help find it. Note that false positives do not necessarily reduce usefulness of the ReqMon: even when the false positive rate was 7%, the effort savings was still over 99%, because nodes need only be inspected until the first true violation is found, which is likely to happen long before most false positives are inspected.

5 Related Work

Various validation tools with a visualisation component to help identifying behavioural problems in the system have been proposed[3, 10, 11, 4, 12, 13, 15]. However, to the best of our knowledge, none of these approaches focuses on the problem of personal requirements. All require complete knowledge of the system models.

In [15] the authors propose graphical animation based on Timed Automata to support validation of compositional behavioural models against requirements and presentation of the analysis results to non-technical stakeholders. Similar to our work, the approach allows the behaviour of the system to be interactively visualised by both technical and non-technical stakeholders. However, unlike our work, this approach does not tolerate missing information.

Another approach with well-defined and sophisticated user interfaces that supports simulation and domain specific visualisation is presented in the SCR* tool set[11]. However, SCR supports only concrete simulation in which access to full information of the model and instance data is required. The same restriction of knowing all information about the models applies to SPIN[12] and Statemate[13, 10]. The approach in [4] introduces automatic model driven animation to SCR specifications, from which scenarios are derived automatically. Behaviours represented as goals are animated according to requirements.

The interactive simulation in Design/CPN[1] (coloured petri-nets), in which stakeholders can choose between enabling transitions and bindings, set breakpoints, and ask for more detailed information, resembles our tool. However, although the current version can handle a very large state space (more than 100,000 nodes), GSTVIEW's dynamic deduction of connections supports systems with millions of nodes and unbounded state spaces, like the email system.

Symbolic and infinite state model checkers[2] are capable of symbolic reasoning and hence can handle missing information symbolically. However, because of their non-interactive nature, they do not provide all the advantages for high level personal requirements validation that symbolic behavior browsing does. Such tools are also incapable of dynamically deducing relevant nodes; thus, they cannot handle large scale systems like email.

Li et al[14] approach validation by using a model checker based on 3-valued logic (based upon the work of Bruns and Godefroid[1]) to prove rich interface descriptions for modular components. Then, checking whether a property holds true of a composition of features is relegated to a light-weight check instead of a full model checking run over the composed system. In their work, there is no emphasis placed on the need for (nor capability for) inspection of behaviors. Similarly, they have no equivalent to our notion of approximate formalization, including such notions as Noticers (which only point out potentially interesting situations to be inspected). Their approach does provide a limited tolerance of missing information, in that it can accommodate missing control information (such as whether a feature is active) by checking both possibilities. In this respect, it is similar to when GSTView provides ReqMon feedback on sibling tree branches. However, unlike our approach, their work does not handle missing "data-like" information, such as the list of valid users of a service or the value of an encryption key. When such information is missing, their answer is likely to be simply "Unknown", whereas our approach replaces missing information with symbolic placeholders that can appear within functional expressions, from which the user is often able to extract meaningful validation results. This difference arises because GSTView is designed to handle function rich models, while their tool focuses on Boolean models. Another important difference between our approaches, however, is that they fundamentally assume that relevant properties can be anticipated and formalized in advance. Not only do new features and components introduce new concepts into the domain's logic, but new *stakeholders* bring new sets of concerns at composition and use time. GSTView is specifically designed to accommodate dynamically changing, idiosyncratic and even unformalizable personal requirements.

Robinson's approach[17] uses automated checkers for requirements applied to actual system traces (not on system models as in our system). It can support large systems and approximate checkers, but requires actually running an instrumented version of the system, so doesn't provide the

same help in avoiding risky transactions altogether.

Hall[7] has proposed a methodology for discovering feature interactions in distributed systems like email. He applied it to discover 27 undesirable behaviors among 156 scenarios examined. His approach assumes complete information about the system being validated and is not supported by automated requirements checkers. Our email study was based on the same feature models as that study, but we did not assume complete knowledge of instance data. For the scenarios we considered, many of the interactions Hall found manually were found in our system by the ReqMons.

6 Conclusion

The ASBB approach and GSTVIEW tool are the first to take on directly the problems of validating high level personal requirements within large and incompletely known distributed systems. Of course, there are limitations, many of which suggest future work. Perhaps the most important include providing support for formalizing requirements approximations and high quality theories in general; improving the usability of the GSTVIEW tool itself in several ways; and continuing to study and validate the ideas in this paper in other domains. A clear limiting assumption is that the nodes of the distributed system have models; the OPENMODEL project is targeted toward building an infrastructure for publishing and retrieving models generated from many modeling tools over the net. Finally, GST size is a clear concern as well; the trees must be large enough to exhibit requirement violations, yet not so big that the user cannot reasonably inspect them. This places a minimum constraint on how much information must be present. As we saw in the BOOKFIND study, a completely unspecified input resulted in doubling the GST size; in the EMAIL study, the unspecified input actually reduced the tree to a small but virtually useless size that did not exhibit behaviors of interest. Dealing with this issue is an open question. One approach to handling large trees we are pursuing is to have the tool detect *similar subtrees* and replace copies of subtrees with icons to avoid duplicate inspection work.

Even given these limitations, however, we believe the present paper has contributed both in drawing attention to assumptions **A1-A4** (and the need to deal with systems that violate them), as well as outlining the design of an approach and tool that can address the problems of large incompletely known systems. The evaluations of Section 4 provide support for our claims that symbolic behavior browsing provides important advantages in personal requirements validation, and that approximate formalization of high level requirements can significantly reduce validation effort.

References

[1] G. Bruns and P. Godefroid. Generalized model checking: Reasoning about partial state spaces. In *Proc. Intl. Conf. on Concurrency Theory – LNCS 877*, pages 168–182. Springer Verlag, 2000.

[2] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.

[3] Design/cpn – a computer tool for colored petri nets. `www.daimi.au.dk/designCPN/`.

[4] A. Gargantini and E. Riccobene. Automatic model driven animation of scr specifications. In *Proc. Fundamental Approaches to Software Engineering (FASE'03)*, 2003.

[5] R. Hall and A. Zisman. Overview of openmodel-based validation with partial information. In *Proc. 18th Conf. on Automated Software Eng*, 2003.

[6] R. Hall and A. Zisman. Omml: A behavioural model interchange format. In *Proc. 2004 Intl. Conf. on Requirements Engineering*, 2004.

[7] R. J. Hall. Feature interactions in electronic mail. In *Feature interactions in telecommunications and software systems VI*. IOS Press, 2000.

[8] R. J. Hall. Open modeling in multi-stakeholder distributed systems: Model-based requirements engineering for the 21st century. In *Proc. 2002 ISR Workshop on State of the Art in Automated Software Engineering*. UC Irvine ISR, 2002.

[9] R. J. Hall. Specification, validation, and synthesis of email agent controllers: a case study in function rich reactive system design. *Automated Software Engineering*, 9:233–261, 2002.

[10] D. Harel. Statecharts: a visual approach to complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[11] C. Heitmeyer, J. Kirby, B. Labaw, and R. Bharadwaj. Scr*: A toolset for specifying and analyzing software requirements. In *Proc. 10th Computer-Aided Verification Conf*, 1998.

[12] G. Holzmann. *Design and validation of computer protocols*. Prentice Hall, 1991.

[13] Statemate website. `www.ilogix.com`.

[14] H. Li, S. Krishnamurthi, and K. Fisler. Modular verification of open features through three-valued model checking. *Automated Software Engineering*, (to appear).

[15] J. Magee, N. Pryce, D. Gyannakopoulou, and J. Kramer. Graphical animation of behavior models. In *Proc. 22nd Intl. Conf. on Software Eng.*, 2000.

[16] Openmodel home page. `www.research.att.com/~hall/openmodel-project.html`.

[17] W. Robinson. Monitoring web services requirements. In *Proc. Eleventh IEEE Intl. Requirements Engineering Conf*, pages 65–74. IEEE Computer Society, 2003.