A Graphical Interval Logic for Specifying Concurrent Systems*

L. K. Dillon, G. Kutty, L. E. Moser
P. M. Melliar-Smith and Y. S. Ramakrishna

Departments of Computer Science and
of Electrical and Computer Engineering
University of California, Santa Barbara 93106

Abstract

The paper describes a graphical interval logic that is the foundation of a toolset supporting formal specification and verification of concurrent software systems. Experience has shown that most software engineers find standard temporal logics difficult to understand and to use. The objective of this work is to enable software engineers to specify and reason about temporal properties of concurrent systems more easily by providing them with a logic that has an intuitive graphical representation and with tools that support its use. To illustrate the use of the graphical logic, the paper provides some specifications for an elevator system and proves several properties of the specifications. The paper also describes the toolset and the implementation.

1 Introduction

One of the great challenges facing today's software engineers is the development of correct programs for real applications. Recent advances in hardware reliability and fault tolerance technology can assure extremely low hardware failure rates for devices. Unfortunately, technologies for digital hardware design and software engineering have not matched this advance. The use of computers in many critical applications is now primarily limited by the reliability of system designs and implementations.

The most critical real applications often involve concurrency, which increases the difficulty of system development and validation. Modern methods of structured programming, which are quite

*Research partially supported by NSF grant CCR-9014382 with cooperation from DARPA. An early version of the paper was presented at 14th Inter. Conf. Software Engineering, May 1992 (Institution of Engineers Australia, IEEE Computer Society, Association of Computing Machinery, Institution of Radio and Electronic Engineers Australia, and Australian Computer Society).

effective for sequential programs, are notoriously inadequate for concurrent ones. Moreover, the nondeterminism inherent in applications that involve concurrency and the reactive character of those applications makes them hard to test. Aggravating these problems is the need to explore large spaces of possible executions, which grow exponentially with the number of independent threads of control.

Formal methods for specifying and verifying systems can, in principle, offer a greater assurance of correctness than informal design/code checks or testing. Formal verification methods can demonstrate that a high-level design meets formally specified correctness requirements, thereby reducing the risk that faulty designs will be used as the basis for system development. Formal specifications are valuable for defining interfaces between independently developed software modules and for establishing software and interface standards. Because they provide a succinct and unambiguous statement of system requirements, formal specifications can potentially be analyzed for consistency, a particularly difficult and important problem for concurrent systems. Formal specifications can also be used during the selection of test data to suggest behaviors that should be tested and, later, to determine whether the execution of a test case is correct or erroneous. Thus, system developers can use formal specifications throughout the system lifecycle to guide development, maintenance and enhancement.

In practice, however, system developers seldom make significant use of formal specification and verification methods. We believe that this is due, in large part, to the reliance of those methods on mathematical formalisms that are difficult to understand and to use. Formal specification and analysis methods must be made accessible to system designers and software engineers if they are to be used in the development of real world systems. Users must be able to express the properties of the systems about which they wish to reason as naturally as possible and to confirm mechanically that the specifications, designs, testing criteria and sample executions have the required properties.

Temporal logics [2, 19, 21, 32] are well-suited for specifying temporal properties of concurrent systems. Experience has shown, however, that specifications of even moderate-sized systems are too complex to be readily understood. This complexity stems chiefly from the need to establish the temporal context within which properties, such as bounded liveness and invariance, must hold. Interval logics [11, 31] address this problem by defining temporal intervals to represent such contexts. For example, to express the requirement that a process that releases a lock on a database must signal that it intends to enter the database before obtaining a new lock, an interval might be used to represent the activity of the system from the time that the process releases the lock until it acquires a new one; the process would then be required to signal its intension within the restricted context represented by the interval (bounded liveness).

Stylized pictures often show complex timing relationships and dependencies more clearly than linear textual representations of the same information. Such diagrams correspond more closely to common conceptualizations than does linear text. Software engineers often draw timing diagrams, like those used to denote signal levels in hardware designs, when describing and reasoning about properties of systems. Even logicians who are fluent in temporal logic find timing diagrams helpful to explain the meanings of temporal logic formulas and to motivate lines of reasoning (see for example [6]). However, because timing diagrams often lack a formal semantics, they cannot be used for rigorous analysis of system properties. Pictorial documentation is all too often ad hoc and liable to ambiguous interpretation.

This paper describes a visual temporal logic in which formulas resemble the informal timing diagrams familiar to designers of hardware systems and to software engineers. Graphical Interval Logic (GIL) has a formal model-theoretic semantics and is more expressive than propositional temporal logic with Until and without Next [27]. It thus provides an intuitive and natural visual notation in which to express system specifications without sacrificing the benefits of a formal notation. A visual editor allows GIL specifications to be easily constructed and to be stored in and retrieved from files. The editor also provides a visual interface to a proof checker and model generator, which permit verification of temporal inferences.

The paper first provides an overview of GIL in Section 2. It then presents sample specifications for an elevator system in Section 3 and shows, in Section 4, how a designer uses the specifications to reason about properties of the system. Section 5 describes the GIL toolset and Section 6 provides a high-level overview of the implementation. Related work is discussed in Section 7, with conclusions and future work presented in Section 8. The appendices provide a model-theoretic semantics for the logic and details of several proofs used in Section 4.

2 Graphical Interval Logic

When reasoning about temporal properties exhibited by a concurrent system during a computation, it is convenient to regard the system as passing through a sequence of states. To model a non-terminating computation, the state sequence must be infinite. A terminating computation can also be modeled with an infinite state sequence by repeating, or *stuttering*, the final state. This permits a concurrent system to be identified with the set of infinite state sequences that represent its potential computations. GIL specifications for a system describe properties of legal state sequences. That is, the specifications must hold at the first state of every infinite state sequence that represents a computation of the system. We adopt a total order model of computation, rather than a partial order model, which has some advantages for representing causality in concurrent systems [26], because total orders are more readily abstracted into meaningful "intervals" that can be represented pictorially at an appropriately high level.

A GIL formula is evaluated at a state in an infinite sequence of states. Infinite state sequences, therefore, provide the *contexts* within which formulas are evaluated. A formula that holds at a state in a context describes a property of the state's *future* within the context, or of the infinite tail sequence that begins with the state and extends through the end of the context. A reflexive interpretation of the future, in which the future includes the present, allows a semantics that is insensitive to finite stuttering. This facilitates the use of hierarchical abstraction and refinement while reasoning about concurrency [18].

Intervals permit the specification of contexts within which properties hold. We denote an interval by a left-closed right-open line segment: $[\!\!-\!\!-\!\!-\!\!-)$. An interval thus shows the individual states in a context as points on a line segment, with the horizontal dimension showing progression through the context (time progresses from left to right). As suggested by the representation, every interval has an initial point (state). However, since contexts are infinite, we do not regard an interval as having a final point.

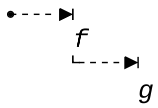
Interval formulas are the heart of the graphical interval logic. The basic interval formula asserts that a formula holds at the first state of a designated context (interval). Derived operators make it possible to assert that a formula holds at every point in a context or at some arbitrary point in a context. GIL therefore also provides the usual temporal Eventually and Henceforth operators.

The logic provides two search primitives for use in specifying intervals.

- Search to a target formula f, represented $\bullet\!\dashrightarrow\!|_{f}$
- Search to the right end of the context, represented $\bullet\!\dashrightarrow\!|$

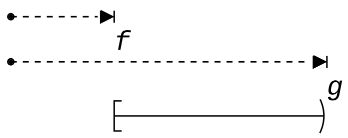
A search to a target formula locates the first future point at which the target holds. The dot represents the point at which the search starts. The search fails if the target does not hold at any future point (inclusive of the present) in the context. A search to the end of the context permits the specification of tail intervals. It can be viewed as locating the right end of the context and, therefore, never fails.

Searches can be composed sequentially, with each successive search starting from the point, if any, located by the previous search. For example,



locates the first point at which f holds and then, beginning from this point, locates the next point at which g holds. We allow the shorthand notation $\bullet\dashrightarrow|_{f}\!\dashrightarrow|_{g}$ when the target f of the first search is a state formula, which does not involve any temporal operators. Because this shorthand produces more compact formulas, we use it extensively in the examples below.[1] The search $\bullet\dashrightarrow|_{f}\!\dashrightarrow|$ is equivalent to $\bullet\dashrightarrow|$, provided that f holds at some future point; however, if the search $\bullet\dashrightarrow|_{f}$ fails, then $\bullet\dashrightarrow|_{f}\!\dashrightarrow|$ also fails. A tail search, when it appears, must be the last search in a search pattern.

The extent of an interval is specified by means of a pair of search patterns, which designate the searches needed to locate the left and right ends of the interval. Both searches begin from the same point. We therefore draw them one beneath the other, with their start points horizontally aligned. The interval determined by the searches is drawn directly beneath the searches, its left end horizontally aligned with the point located by the first search pattern and its right end horizontally aligned with the point located by the second search pattern. For example:

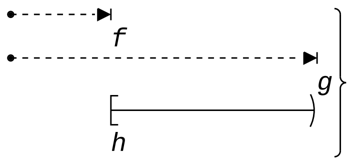


The interval starts at the point located by the search for its left end and extends up to, but does not include, the point located by the search for its right end. The above diagram thus represents the interval that starts with the first point at which f holds and ends just prior to the first point at which g holds. The interval cannot be constructed if either search fails or if the interval is empty, meaning that the point specified by the first search pattern does not precede that specified by the second search pattern.

[1] As a result, the examples are slightly less general than they might be, since a sequence of searches cannot always be drawn on the same line. However, they are easily converted into more general examples by drawing each primitive search on a new line.

Figure 1 illustrates conventions that simplify the representation of several different types of intervals, which occur frequently in specifications of concurrent systems. The first abbreviation, in which a single search specifies the extent of an interval, is permitted when the search for the interval's left end is a prefix of the search for its right end. Thus, the interval in the first example begins with the first point at which f holds and extends up to, but does not include, the next point at which g holds. The interval cannot be constructed if f does not hold at any future point, if g does not hold at any point in the future of the first point at which f holds, or if g holds at the point located by the search to f. The second example in Figure 1 is a special case of the first, in which the target of the first search is $true$. A search to $true$ succeeds immediately, locating the point at which the search begins. The interval in the example thus begins with the point at which the specification is evaluated and extends up to the next point (exclusive) at which f holds. The interval cannot be constructed if f does not hold at any future point or if f holds at the present point. Such intervals are used to specify finite prefixes of larger contexts. The triangle $\triangle$ in the third example is called the point operator. As illustrated by the example, the point operator appears directly below the point located by the final search in a sequence of searches and constructs the tail interval that starts with the point so located. The point operator is used to locate a point within a context and, when the point is located, assert that a property holds at that point. The point cannot be located if any of the searches fails. The final example shows that, by itself, an interval line represents the full context.

To assert that a formula h holds at the initial point of an interval, h is drawn left justified below the left delimiter. For example,



(1)

asserts that h holds at the first point of the designated interval. A formula holds vacuously at the first point of an interval that cannot be constructed. Thus, if either f or g never holds in the future, or if the first (future) point at which f holds does not precede the first (future) point at which g holds, then (1) holds by default. The right brace helps, visually, to delimit the formula. The subformulas that appear in an interval formula (e.g., f, g and h in the above formula) may be general graphical interval logic formulas.

GIL provides the usual logical operators: $\wedge$ (conjunction), $\vee$ (disjunction), $\Rightarrow$ (implication),

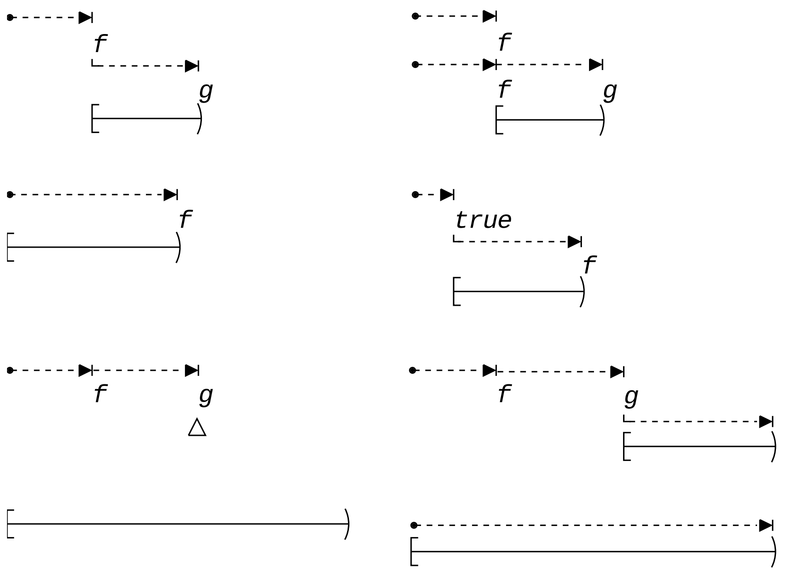


Figure 1: Examples of some derived intervals (left column) and their definitions (right column).

$\equiv$ (equivalence) and $\neg$ (negation). Formulas composed of subformulas that contain intervals are drawn using a vertical layout. In vertical layout, the operands of a binary operator are left justified, with the first operand above the second and the operator between them, and a formula to be negated is drawn left justified below the negation sign.

$$\begin{array}{llllll} f & f & f & f & & \\ \wedge & \vee & \Rightarrow & \equiv & \text{and} & \neg \\ g & g & g & g & & f \end{array}$$

Conjunction is the default in vertical layout, so that the operator $\wedge$ can be omitted in the first example above.

Both layout and precedence rules determine the grouping of operations. GIL formulas obey a variation of Landin's offside rule [20], which requires that every token of a formula lie in the lower right quadrant determined by the upper left corner of the smallest rectangle that contains its first token. The first token that does not obey this rule, called an offside token, terminates the parse of a formula. The precedence of operators (from high to low) is: negation, conjunction, disjunction, implication, and equivalence. Binary operators associate from left to right. Right braces delimit interval formulas and permit explicit grouping of operations.

The weak Until operator U of propositional temporal logic (PTL) is expressed in GIL as follows.

(2)

The formula asserts that g holds at the first point where either f does not hold or g does hold, unless no such point is located. In the later case, f (as well as $\neg g$) holds at all future points. In other words, f holds at least until g holds, which is the usual semantics of f U g.

GIL provides a special syntax for expressing invariants and eventualities. To assert that a formula holds at every point in an interval, the formula is drawn indented directly below the interval. To assert that a formula holds at some point within an interval, the formula is drawn left justified directly below a diamond $\diamond$ drawn on the interval. Figure 2 shows these conventions and their definitions. The definition (top right) of the invariant notation (top left) can be understood as follows. Since *false* does not hold at any point of a context, the point formula holds precisely if the search to $\neg f$ fails, that is, precisely if f holds at all future points. This definition is consistent with the usual temporal identity $\Box f \equiv f$ U *false* and with the GIL formula (2) for f U g. The formula that defines the eventuality notation holds precisely if the search to f succeeds, that is, precisely if f holds at some future point.

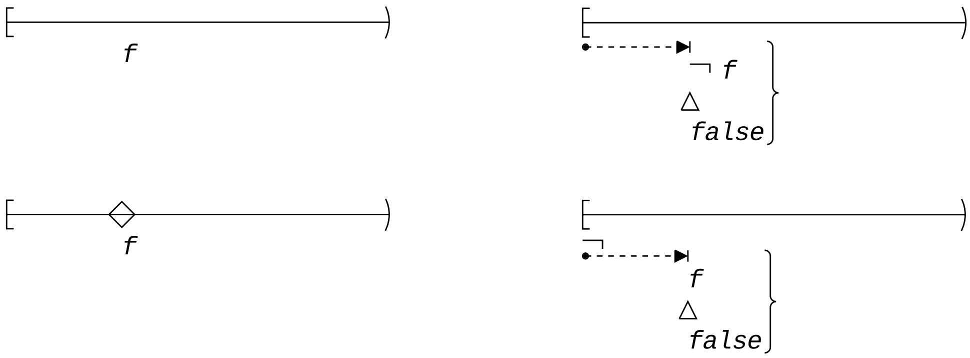
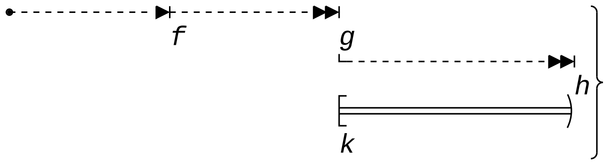


Figure 2: Representation of invariants (top left) and eventualities (bottom left), and their definitions (right column).

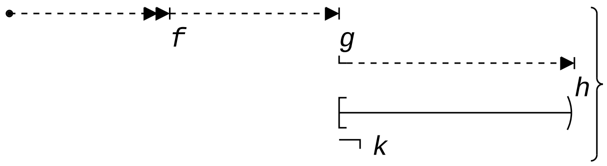
As noted above, an interval formula holds vacuously if any search performed in locating the ends of the interval fails or if the interval is empty. An interval formula is, therefore, implicitly predicated on locating all search targets and on locating the left end of the interval before locating the right end. For this reason, we refer to the search and interval operators described above as *weak* operators. GIL also provides *strong* versions of these operators, which are useful in specifications and in expressing the negations of interval formulas. A strong search, denoted by a double arrowhead, asserts that the search succeeds unless some prior weak search fails. A strong interval, denoted by a double line, asserts that the point located by the search for the interval's left end precedes that located by the search for its right end unless some search fails. For instance,



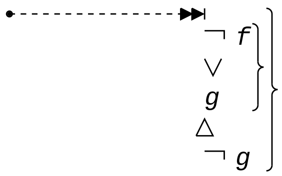
(3)

holds by default if the search for f fails. However, if this search succeeds, then the formula requires that the subsequent searches for g and h succeed, that the interval is not empty (h does not hold at the point located by the second search), and that k holds at the first state of the interval. The appendix gives formal definitions of strong intervals and strong searches.

The dual of an interval formula is obtained by changing the senses (strong to weak and weak to strong) of the interval modality and of the searches for the ends of the interval. This dual relationship implies that negation can be moved into an interval formula by changing the senses of the interval and of its searches. For instance, the negation of (3) is equivalent to



A tail interval, such as constructed by the point operator, is not affected by the sense of the interval modality. Negation can therefore be moved into a point formula by changing the senses of the searches. The negation of (2), for example, is equivalent to



3 An Example Specification

We present a GIL specification for an elevator system to illustrate the ideas in the previous section. The example includes specifications of basic safety and liveness requirements, and also of more complex fairness requirements.

For simplicity, we consider an elevator with three floors. The specification makes use of the following state predicates, for $n = 1, 2, 3$. The predicate *at\$n* is *true* when the elevator is at floor n and *false* when it is not. The predicate *goingup* models a physical switch whose setting, when the elevator leaves a floor, determines the direction of travel: up, if *goingup* is *true*, and down, if *goingup* is *false*. The predicate *open\$n* is *true* when the doors to the elevator are open at floor n and *req\$n* is *true* when there is an outstanding request for service at floor n. Finally, when the elevator is at the second floor, *arriveup* indicates whether it was going up or down when it arrived.

Specifications are read from top to bottom and left to right. By convention, we begin each specification with a context line, which represents a legal execution of the system. The first specification expresses initial requirements and the remaining specifications describe system invariants. We associate labels (shown in bold) with specifications for reference purposes below.

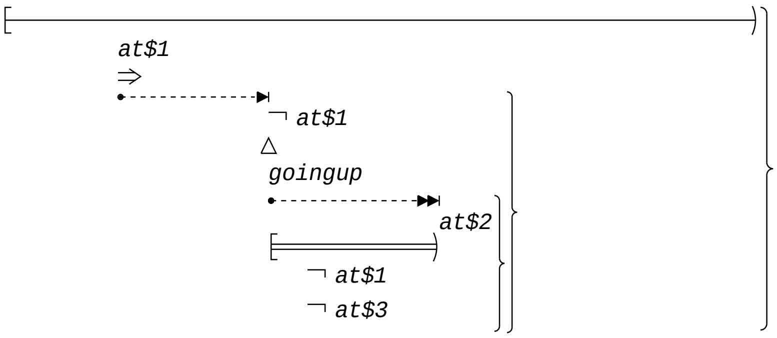
Init. The elevator begins operation at the first floor, all doors are closed, and there are no requests for service.

at\$1
$\neg$ *req\$1*
$\neg$ *req\$2*
$\neg$ *req\$3*
$\neg$ *open\$1*
$\neg$ *open\$2*
$\neg$ *open\$3*

At\$$n$**\$**m, $1 \leq n < m \leq 3$. The elevator is never at two different floors simultaneously.

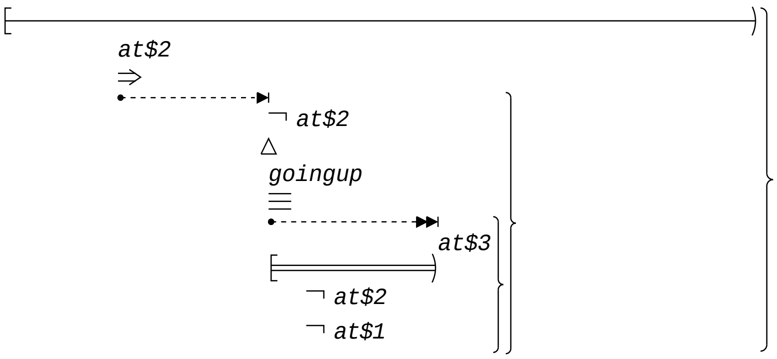
at\$n
$\Rightarrow$
$\neg$ *at\$m*

UpFrom$1. The elevator goes up when it departs the first floor, arriving at the second floor without first visiting any other floors.

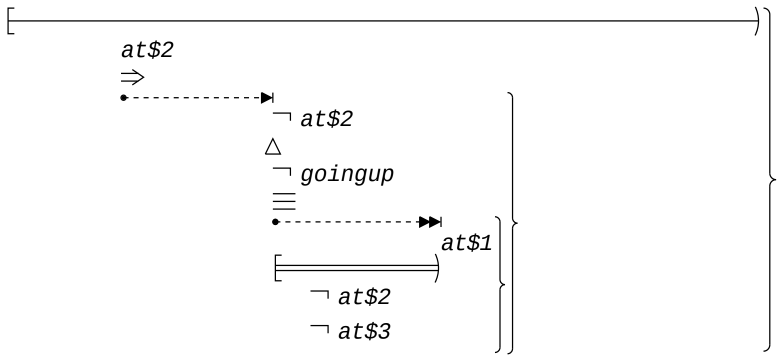


The invariant in this formula is predicated on locating a point at which the elevator has just left the first floor. The specification asserts that the elevator is going up at every such point and that it reaches the second floor before either of the other floors. The strong search requires that the elevator eventually arrives at the second floor and the strong interval requires that it does not arrive there immediately upon leaving the first floor, but takes some time to do so.

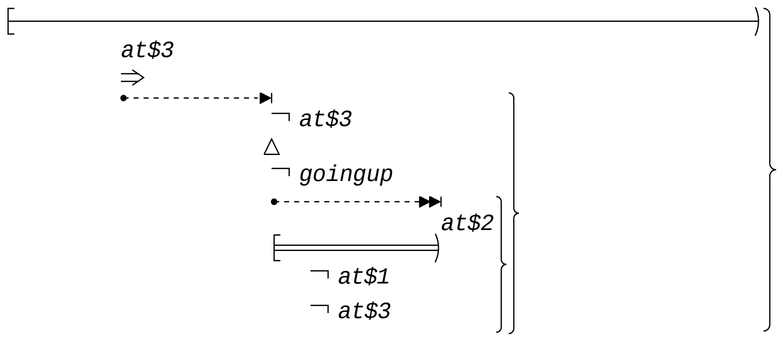
UpFrom$2. The elevator goes up when it departs the second floor precisely if it goes directly to the third floor.



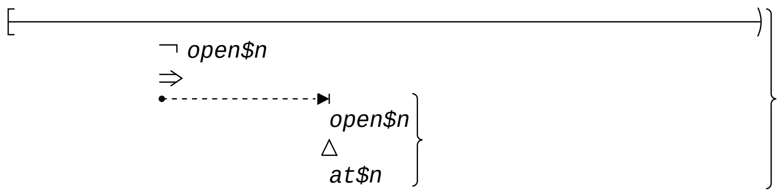
DownFrom$2. The elevator goes down when it departs the second floor precisely if it goes directly to the first floor.



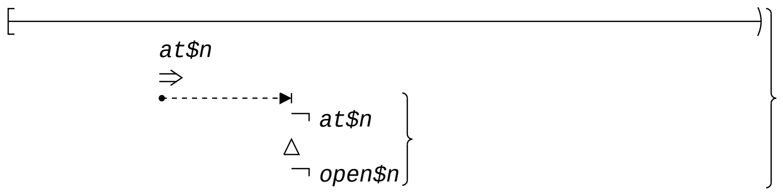
DownFrom$3. The elevator goes down when it departs the third floor, arriving at the second floor without first visiting any other floors.



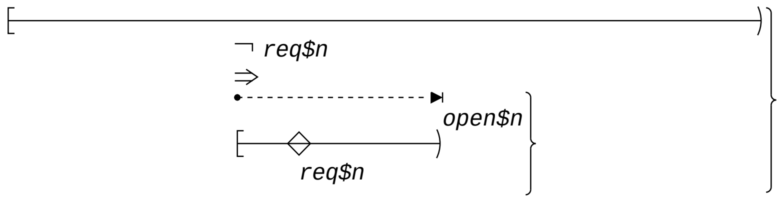
SafeOpen$$n$, $n = 1, 2, 3$. The doors open at a floor only when the elevator is at the floor.



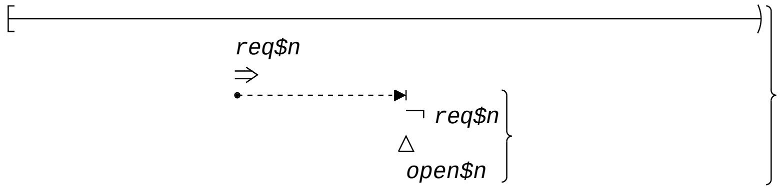
SafeDepart$$n$, $n = 1, 2, 3$. The elevator departs a floor only when the doors at the floor are closed.



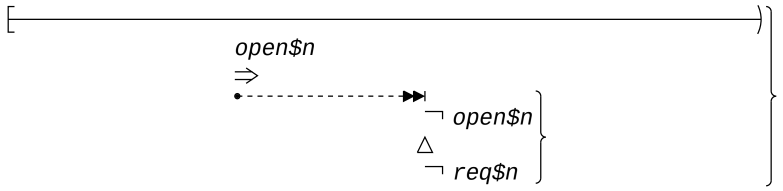
ReqService$n, $n = 1, 2, 3$. The doors open at a floor only in response to a request for service at the floor.



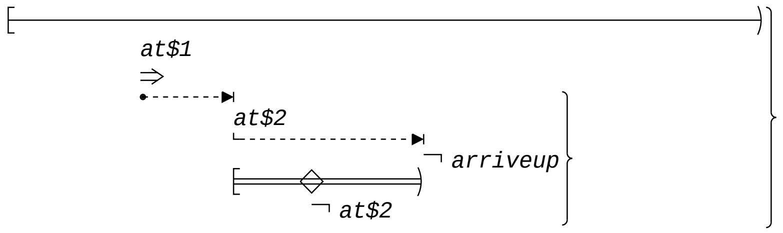
WaitService$n, $n = 1, 2, 3$. A request for service at a floor is only canceled if the floor is being serviced (the doors are open).



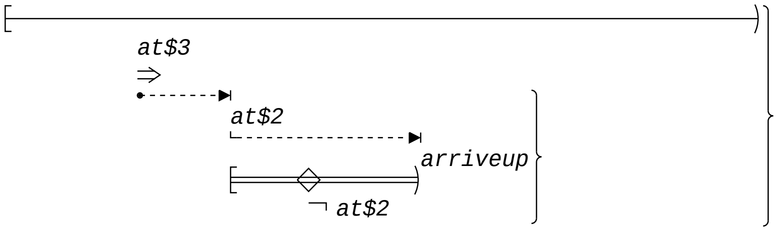
CancelService$n, $n = 1, 2, 3$. The doors do not remain open indefinitely, and all requests for service at the current floor are canceled when they close.



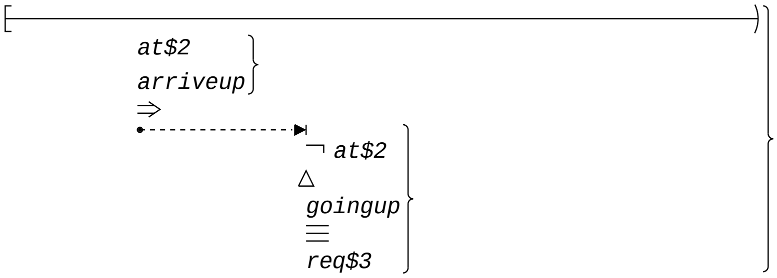
ArriveUp. Whenever the elevator arrives at the second floor from the first floor *arriveup* is *true*, and it remains *true* at least until the elevator departs the second floor.



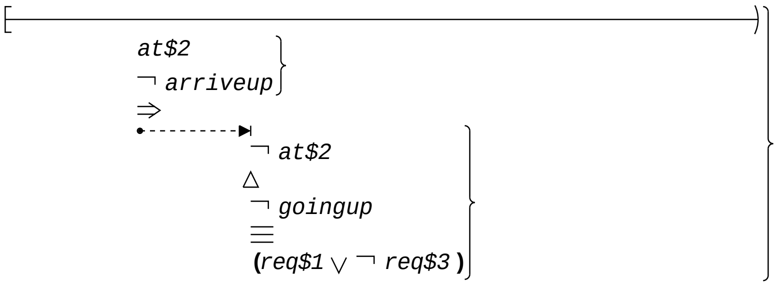
ArriveDown. Whenever the elevator arrives at the second floor from the third floor *arriveup* is *false*, and it remains *false* at least until the elevator departs the second floor.



ContinueUp. If the elevator is going up when it arrives at the second floor, it continues going up when it departs the second floor precisely if someone requires service at the third floor by the time the elevator departs.

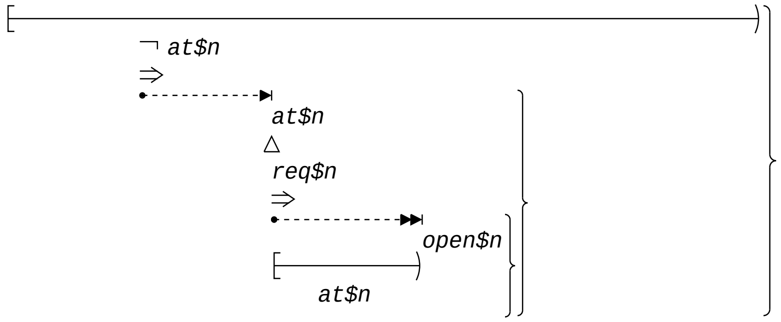


ContinueDown. If the elevator is going down when it arrives at the second floor, it continues going down when it departs the second floor precisely if either someone requires service at the first floor or no one requires service at the third floor by the time the elevator departs.

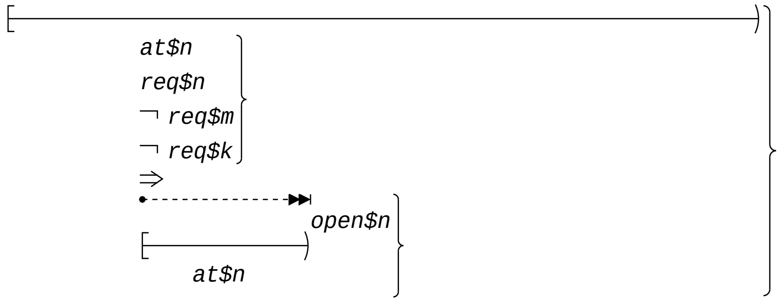


ContinueUp and ContinueDown require that, once the elevator starts traveling in a given direction, it changes directions only if no one requires service at a floor in that direction. The disjunction in ContinueDown permits (but does not require) the first floor to act as the "home floor", to which the elevator can return when it is idle.

**ServeReqsOnArrival$**$n$, $n = 1, 2, 3$. If a passenger requests service at a floor by the time the elevator reaches the floor, the elevator opens its doors before departing the floor.

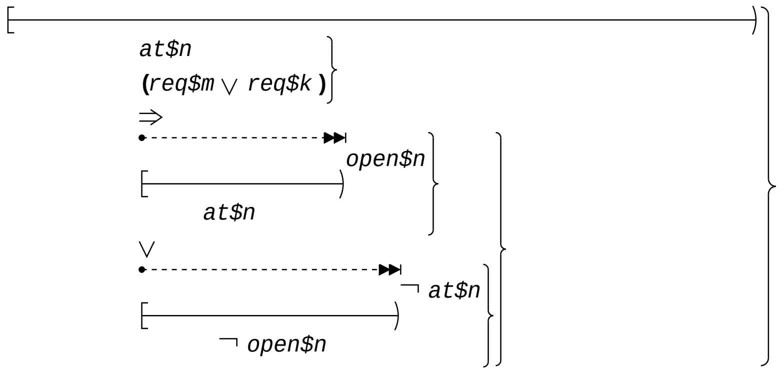


**ServeNoConflict$**$n$, $n = 1, 2, 3$. If a passenger needs service at a floor while the elevator is at the floor and no one needs service at another floor, then the elevator opens its doors before departing the floor.

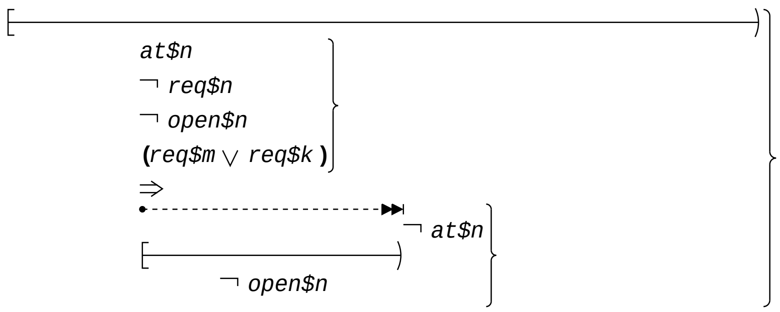


We use m and k in this and the remaining specifications to denote the other floors, that is, $\{ m, k \} = \{ 1, 2, 3 \} \setminus \{ n \}$, and $m < k$.

**ServeOrDepart$**$n$, $n = 1, 2, 3$. If the elevator is at a floor and a passenger requires service at a different floor, then either the doors open at the current floor or the elevator departs the floor (without first opening the doors).



NoServeDepart$n, $n = 1, 2, 3$. The elevator departs a floor without first opening its doors whenever no passenger requires service at the floor, the doors are closed, and someone needs service at another floor.



The last four specifications ensure that the elevator makes progress and that it services floors in a timely fashion. If a passenger requests service at a floor by the time the elevator arrives there, the appropriate ServeReqsOnArrival specification guarantees that the elevator stops at the floor before traveling on to other floors. Similarly, the ServeNoConflict specifications ensure that the elevator services a request if the elevator is at the floor needing service and no one is waiting for service at another floor. However, if a passenger requires the elevator at some other floor, the appropriate NoServeDepart specification ensures that, once the current floor is serviced, the elevator departs the floor without servicing any additional requests for service at the current floor that may be made in the interim. The ServeOrDepart specifications prevent the elevator from sitting idlely at a floor if other floors require service.

The above specifications show how the graphical representation of intervals facilitates the representation of the contexts over which properties hold. Several nested Until operations are typically required to achieve the same effect in PTL. For example, UpFrom$2 and ArriveUp might be expressed in PTL as follows.

UpFrom$2:

$$\Box(\mathit{at\$2} \Rightarrow \mathit{at\$2}\, \mathrm{U}\, [\neg \mathit{at\$2} \wedge \{\mathit{goingup} \equiv (\Diamond \mathit{at\$3} \wedge \neg \mathit{at\$3} \wedge (\neg \mathit{at\$1} \wedge \neg \mathit{at\$2})\, \mathrm{U}\, \mathit{at\$3})]\})$$

ArriveUp:

$$\Box(\mathit{at\$1} \Rightarrow \neg \mathit{at\$2}\, \mathrm{U}\, [\mathit{at\$2} \wedge \{\Box \mathit{arriveup} \vee \neg(\mathit{at\$2}\, \mathrm{U}\, \neg \mathit{arriveup})\}])$$

Horizontal alignment can help, visually, in understanding GIL specifications. For instance, when the antecedent *at$1* of the invariant in UpFrom$1 holds, the alignment of the first search arrow with the antecedent provides a visual reminder that *at$1* holds at the point where the search commences.

Evidentally, then, if the search succeeds, it locates a point at which *at$1* has just become *false*. That *goingup* holds at this point is evident from the positioning of the predicate. Similarly, that *at$2* holds somewhere in the strict future of this point and that *at$1* and *at$3* do not hold until the future *at$2*-point are evident from the positioning of the strong search and the corresponding strong interval.

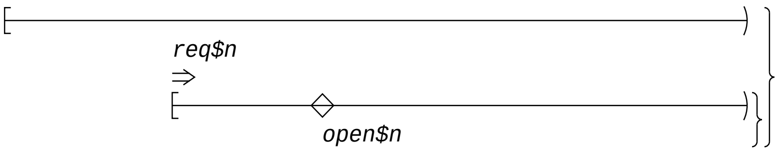
4 Graphical Proofs of System Properties

An important benefit of formal specifications is that they can be analyzed for potential consequences. Analysis can demonstrate that a specification correctly expresses higher-level system requirements and can help the designer learn more about the system under development. If analysis reveals that a specification admits computations that violate requisite properties, this indicates that the specification is incomplete or in error. On the other hand, when desired properties can be proved from the specification, the designer gains confidence that the specification provides a complete and accurate description of the system to be built.

The following are examples of properties that might be required of the elevator system. The first requirement is one of many safety properties that a designer might wish to establish. The second expresses a minimal fairness requirement.

**Safe$**$n$, $n = 1, 2, 3$. The elevator must be at a floor for its doors to be open there.

**Service$**$n$, $n = 1, 2, 3$. The elevator eventually responds to a request for service.



Space does not permit a complete account of the proof of Service$$n$. We therefore give only part of the proof in the body of the paper. The remaining proof appears in Appendix A in the form of intermediate lemmas and annotated proof trees.

The specifications for a system express all temporal constraints on its legal computations. This means that the system satisfies a requirement r if the conjunction s of the specifications implies r

or, equivalently, the implication $s \Rightarrow r$ is valid. In theory, therefore, it suffices to check the validity of this inference using the GIL proof checker to show the requirement is met. In practice, however, theorem proving requires human assistance to be computationally feasible. The designer provides this assistance in our proof method by breaking a complex proof down into inferences that are small enough for the GIL proof checker to validate.

A major advantage of using a visual logic, such as GIL, is that pictures (of the implications) representing the inferences used in the proof of a requirement can show the temporal flow of the argument. The graphical representation of the timeline allows one to align appropriate points in the picture. Such alignment helps the reader see the points at which invariants are being instantiated, the intervals and points being aligned to establish bounded liveness and invariance conditions, the relationships between different points and intervals, and so on. These visual cues can be extremely helpful both for constructing proofs and for discovering potential fallacies. This "syntactic sugar" has no semantic content in the proofs below, although we are investigating a technique that will permit the designer to use alignment to specify orderings of points within a specification.

The alignment and ordering of points on a timeline has other uses as well. For instance, the GIL toolset provides a model generation facility for producing a counterexample in the case that an inference is invalid. The counterexample can be displayed as a sequence of states or as a timing diagram. Aligning the states in the implication appropriately with this counterexample can help illustrate the fallacy in the inference.

The proof of Safe\$$n$ in Figure 3 uses alignment to highlight the underlying correctness argument. The annotations alongside the picture show the specifications used in the proof. As shown by the annotations, Safe\$$n$ is proved from

- Init, which asserts that the doors are not open at floor n when the system starts up
- SafeOpen\$$n$, which asserts that the elevator is at floor n when the doors first open at the floor
- SafeDepart\$$n$, which asserts that the doors have closed by the time the elevator departs the floor

Aligning the invariant in SafeDepart\$$n$ with the point located by the search to *open\$n* in SafeOpen\$$n$ highlights the fact that, when the invariant is evaluated at this point, it guarantees that *at\$n* holds continuously from the (arbitrary) point at which *open\$n* becomes *true* at least until *open\$n* is *false*.

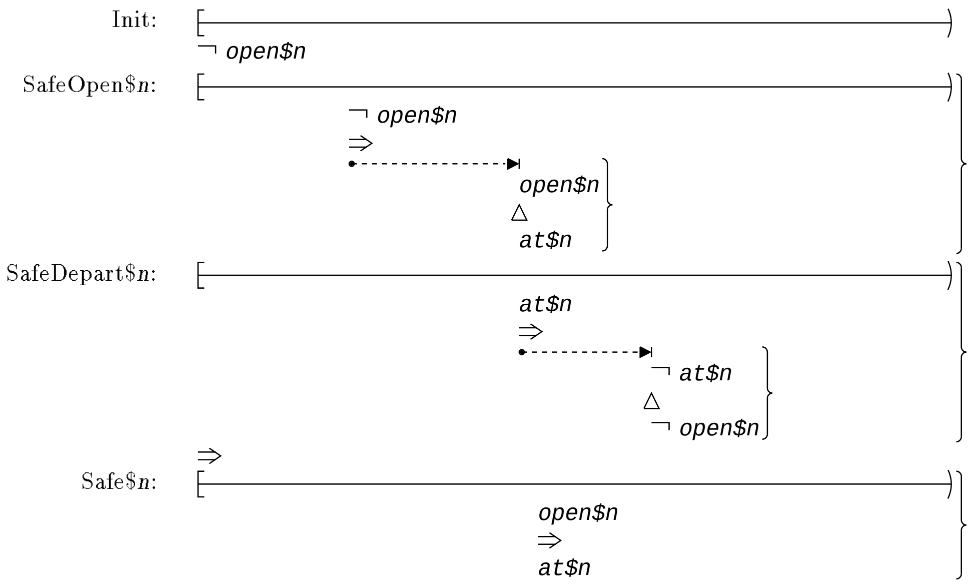


Figure 3: Proof of Safe$n, $n = 1, 2, 3$.

The proof of Service$n is too complex to be accomplished in a single step. Figure 4 shows the last step in the proof. As shown by the annotations alongside the figure, the final deduction makes use of several specifications and of the intermediate result Arrive$n, which is established independently as another step of the proof. The reasoning illustrated by the picture can be understood as follows. If *req$n* holds at a point in a computation, but becomes *false* at a future point, then WaitService$n ensures that the invariant in Service$n holds at this point. To highlight this reasoning, we align the invariants in WaitService$n and Service$n and align the points at which *open$n* is asserted to hold. The remaining premises establish Service$n in the case that *req$n* holds continuously from some point in a computation. We use Arrive$n to deduce that there is a future *at$n*-point. The *at$n*-point is purposely positioned within the span of the search arrow in WaitService$n to remind the reader that we are interested in the case where *at$n* is *true* before *req$n* is *false*. The next three premises represent a case split. The invariant in ServeNoConflict$n establishes the invariant in Service$n in the case that *req$m* and *req$k* are both false at the *at$n*-point. The invariant in ServeOrDepart$n establishes Service$n in the case that *req$m* or *req$k* is *true* at the *at$n*-point and *at$n* holds throughout the future. Finally, the invariant in ServeReqsOnArrival$n when instantiated at the next ¬*at$n*-point, establishes the required invariant in the case that *req$m* or *req$k* is *true* at the *at$n*-point and there is some future point at which *at$n* is *false*.

Figure 5 shows how a complex proof is split into more manageable steps by case analysis.

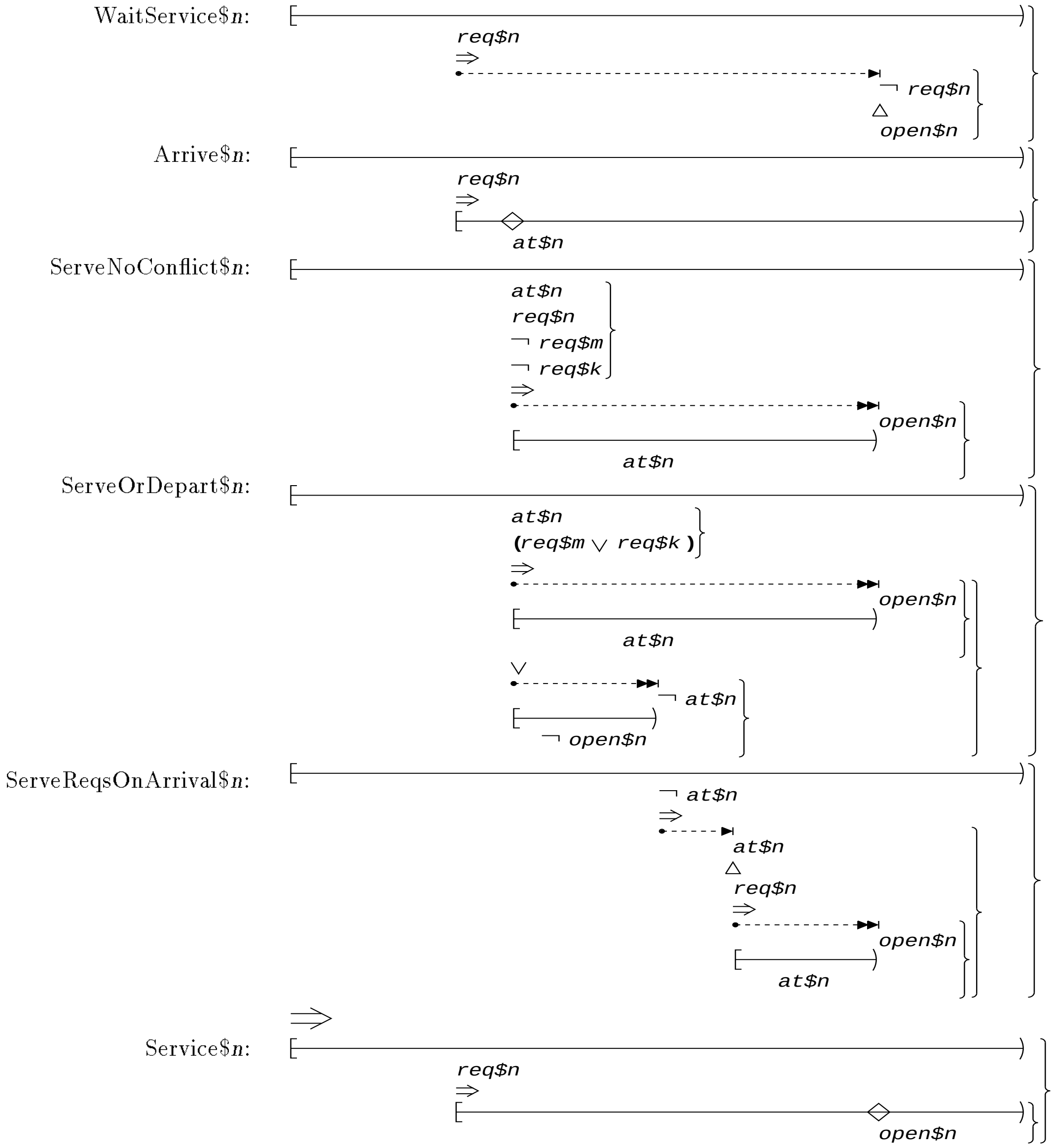


Figure 4: Final deduction in the proof of Service$n, $n = 1,2,3$, $\{m,k\} = \{1,2,3\} \setminus \{n\}$, and $m < k$.

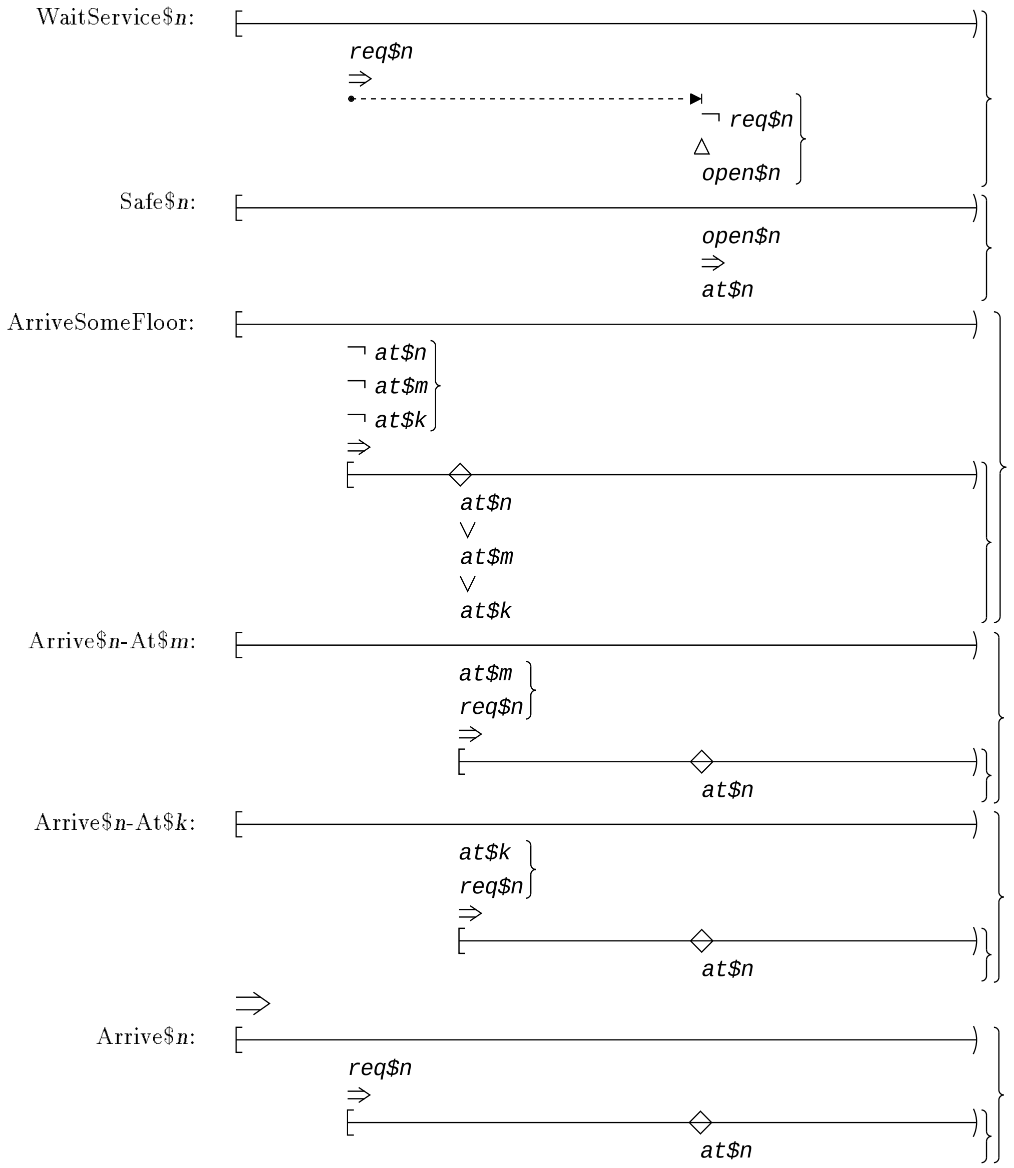


Figure 5: Final deduction in the proof of Arrive\$$n$, $n = 1, 2, 3, \{ m, k \} = \{ 1, 2, 3 \} \setminus \{ n \}$, and $m < k$.

It represents the last step in the proof of Arrive\$$n$. In the same style as the previous example, WaitService\$$n$ and Safe\$$n$ establish the required invariant when *req\$n* is *false* at some future point. The remaining premises are required when *req\$n* holds continuously from some point in a computation. ArriveSomeFloor represents a progress requirement needed to ensure that the elevator does not remain in transit indefinitely, but eventually arrives at some floor. This permits the proof to be reduced to the two cases represented by Arrive\$$n$-At\$$m$ and Arrive\$$n$-At\$$k$, which assert, respectively, that the elevator eventually arrives at floor n from floor m and that it eventually arrives at floor n from floor k.

The requirement ArriveSomeFloor can be proved directly from the specifications Init, UpFrom\$1, UpFrom\$2, DownFrom\$2 and DownFrom\$3. However, the temporal flow of the argument is more readily apparent if the proof is broken into several steps, the final step being verification of the deduction in Figure 6. The premises in the figure follow trivially from the UpFrom/DownFrom specifications and Init.

Proofs of Arrive\$$n$-At\$$m$ and Arrive\$$n$-At\$$k$ are required to complete the proof of Service\$$n$. The high-level strategy used in the proofs is to first show that the specifications ensure the elevator does not remain at a floor indefinitely if it is needed at a different floor. These "departure results" and UpFrom\$1, which guarantees that the elevator eventually arrives at the second floor if it ever leaves the first, ensure Arrive\$2-At\$1. Similarly, the departure results and DownFrom\$3 imply Arrive\$2-At\$3. For the proof of Arrive\$1-At\$2 and Arrive\$1-At\$3, the departure results and the specifications are first used to show that, if the elevator is traveling down when it arrives at the second floor, it eventually arrives at the first floor. Arrive\$1-At\$3 follows easily from this, the departure results, and the specifications. Finally, we use Arrive\$1-At\$3, the departure results and the specifications to show that the elevator eventually arrives at the first floor if it is traveling up when it reaches the second one. The proofs of Arrive\$3-At\$2 and Arrive\$3-At\$1 parallel those of Arrive\$1-At\$2 and Arrive\$1-At\$3. The departure results are established by a straightforward (but tedious) case analysis. The details of the proofs appear in Appendix A.

5 The Graphical Interval Logic Toolset

We recently built a prototype toolset to demonstrate proof-of-concept and permit experiments with the logic. The prototype includes a visual editor that allows specifications to be easily constructed and to be stored in and retrieved from files, a proof checker that mechanically checks the validity

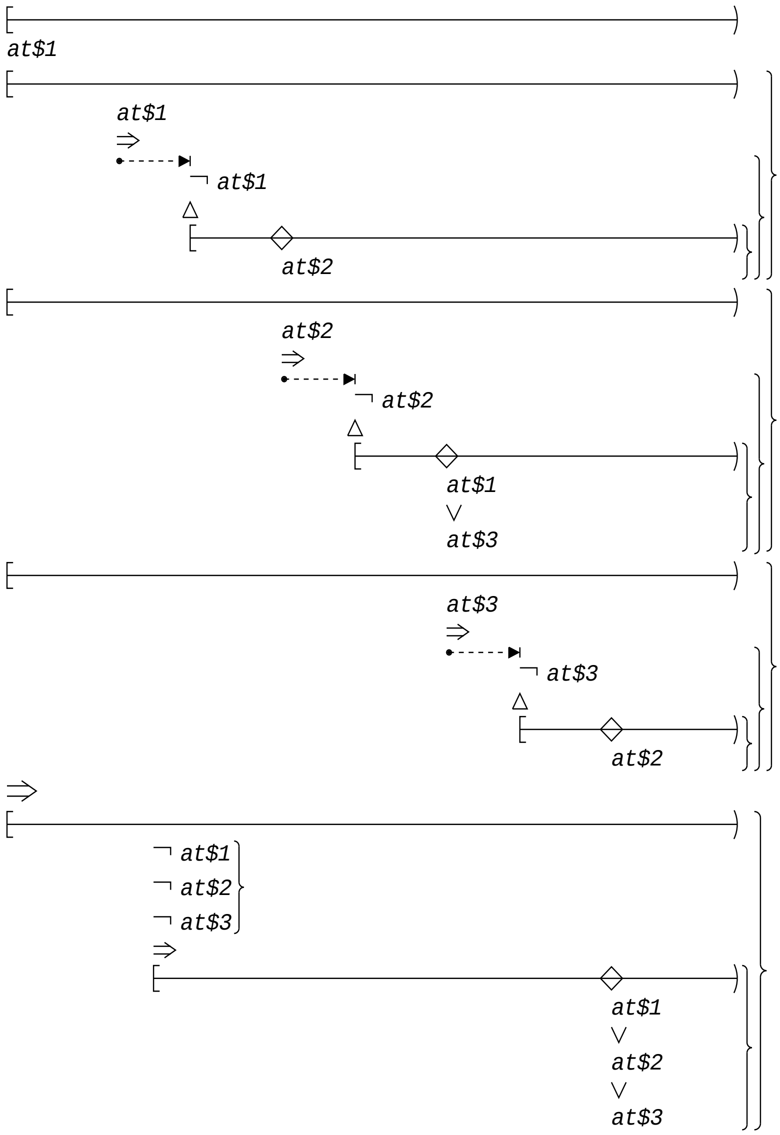


Figure 6: Final deduction in the proof of ArriveSomeFloor.

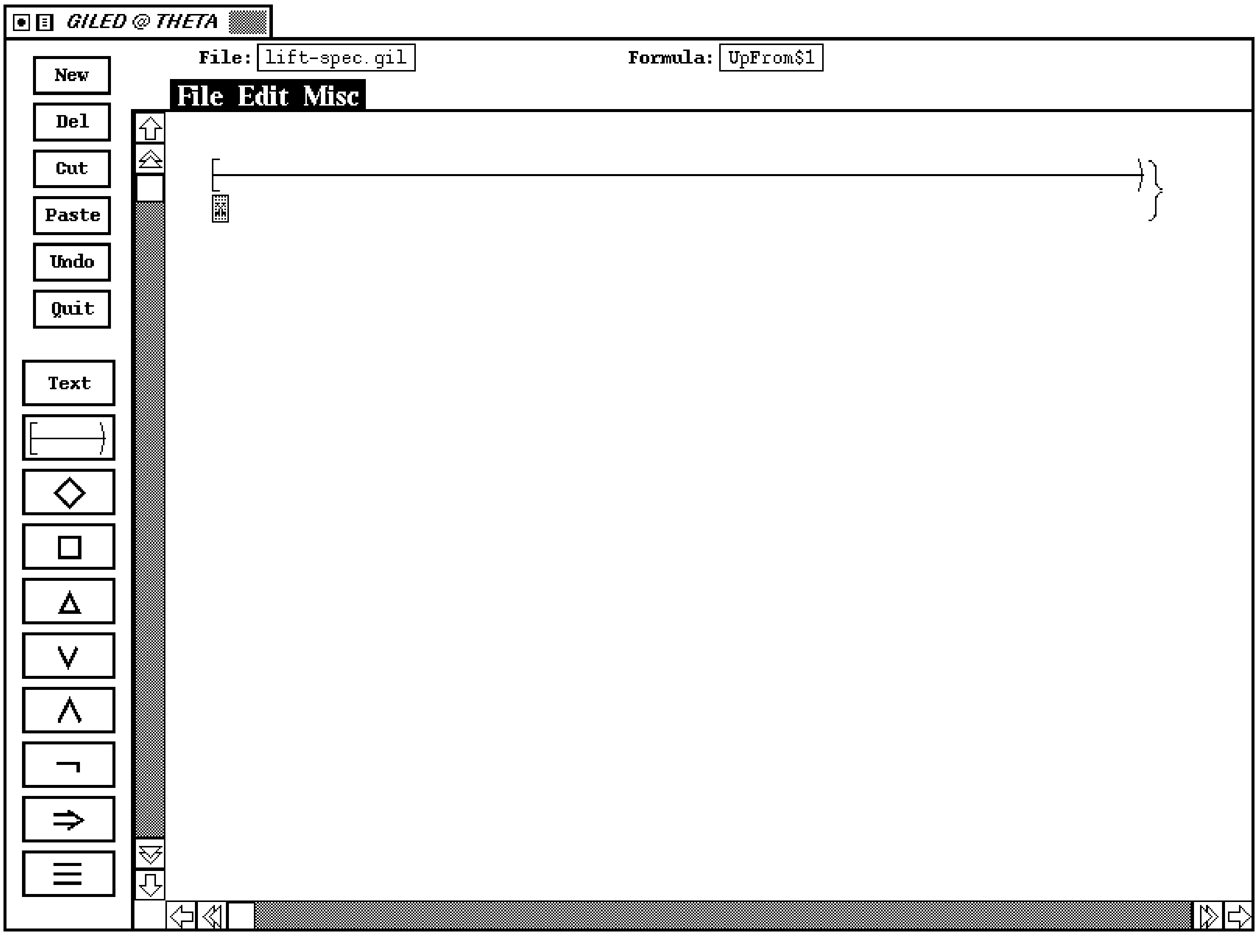


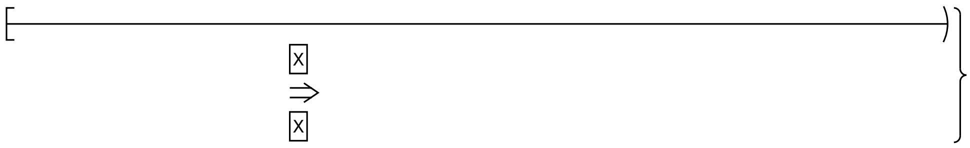
Figure 7: GILED User Interface.

of temporal inferences, and a model generator that exhibits state sequences over which formulas hold. This section provides a brief overview of the GIL toolset.

Figure 7 shows the appearance of the interface of the GIL editor (GILED). Formulas are edited on a canvas, which comprises the main region of the display. The canvas in Figure 7 contains a template for creating a new specification. The template consists of an outer context interval and a box, automatically positioned below the start of the interval, that represents a formula that has yet to be defined. The designer uses the mouse during editing to select formulas in the canvas and editing operations; the box is selected (indicated by shading) in the example. Scroll bars permit the canvas to be scrolled for viewing large formulas.

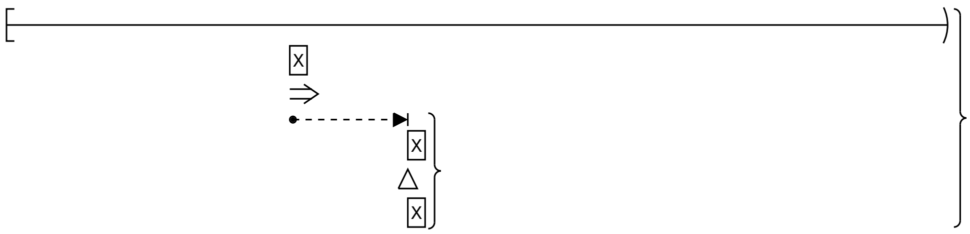
The buttons in the panel on the lower left side of the display correspond to GIL primitives. The `Text` button allows a box to be replaced with a state predicate. The remaining buttons in the lower left panel specify GIL operators that apply to appropriate formulas. First are buttons corresponding to the four temporal operators: the interval [——) , eventuality $\Diamond$, invariant $\Box$, and point $\triangle$ operators.[2] The last five buttons correspond to the propositional operators: disjunction $\vee$, conjunction $\wedge$, negation $\neg$, implication $\Rightarrow$, and equivalence $\equiv$. The buttons in the upper left panel provide language independent editing operations. Commands to override the default layout of formulas and commands for storing and retrieving formulas are found in the `Edit` and `File` pull-down menus. The proof checker is invoked and models are displayed using commands provided in the `Misc` pull-down menu; models are displayed graphically in an accompanying window (not shown in Figure 7).

Briefly, to build the formula UpFrom$1, a designer might begin by selecting the `New` button,[3] which produces the template shown in Figure 7. The $\Box$ and $\Rightarrow$ buttons can then be used to (automatically) indent the box below the context line and expand it into an implication. This produces the following template for an invariant implication.



GILED selects a box to expand next by default; however, the designer may override the default selection at any time using the mouse.

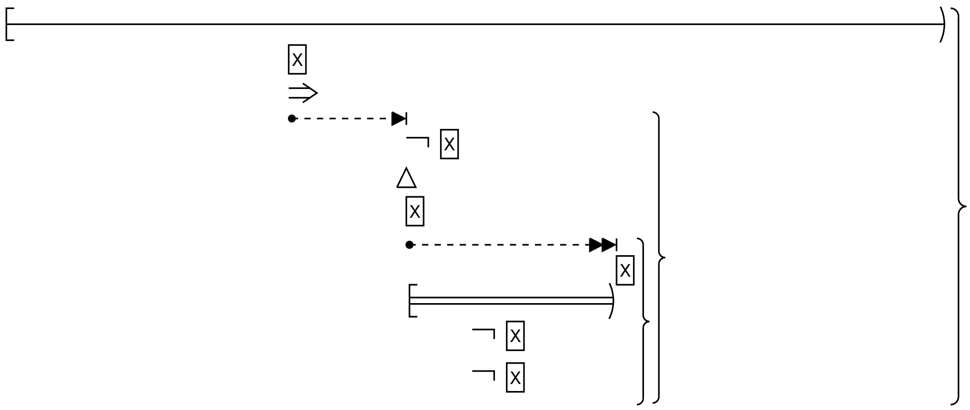
Selecting the second box and the $\triangle$ button converts the consequent into a point formula. For this requirement, the designer uses the mouse to position a single search arrow; GILED then produces a point symbol and a box to represent the search target, as shown below.



[2] As noted in Section 2, the eventuality, invariant, and point operators are derived from the interval operator. However, they correspond to common conceptualizations that are distinguished by the graphical syntax for visualization purposes.

[3] The `New` button in the current implementation of GILED does not automatically generate the right parenthesis. This will be rectified in the next version of GILED.

The designer can continue in this fashion to produce a template with the required structure.



The interval in this template is created by first expanding a box into an appropriate interval template, using the mouse to position the interval and the search arrow. The button produces weak search arrows and weak intervals by default, so that the designer then clicks the mouse on the appropriate search arrow and interval to obtain their strong counterparts. To convert the above template into UpFrom$1, the designer selects the pending boxes in turn, clicks on the `Text` button and types the state predicates.

In addition to the editing operations illustrated above, GILED provides capabilities for cutting and pasting formulas, resizing intervals and search arrows, repositioning invariants and eventualities, and so on. If a formula does not fit in the space alloted, GILED indicates an error and highlights the oversized formula. The designer can correct the error by resizing contexts and searches and repositioning formulas. The editor automatically resizes all affected (sub)formulas to scale.

GILED interfaces with the GIL proof checker and model generator, allowing the designer to work entirely with graphical formula. Functions that access these tools components are provided in the `Misc` pull-down menu under the labels: `Check Proof`, which determines if the formula in the canvas follows from premises designated by the designer; `Prove`, which checks the formula in the canvas for validity; and `Construct Models`, which determines if the formula in the canvas is satisfiable. `Check Proof` keeps track of the structure of each proof and checks for circular reasoning. Once verified, a requirement need only be reverified if the designer modifies premises used (either directly or indirectly) in the proof or if the designer wishes to modify the proof structure. In situations where a proof fails or a formula to be proved is not valid, a counterexample can be displayed in a separate window. Alternatively, `Generate Models` permits a model (infinite state sequence) that satisfies the formula in the canvas to be displayed.

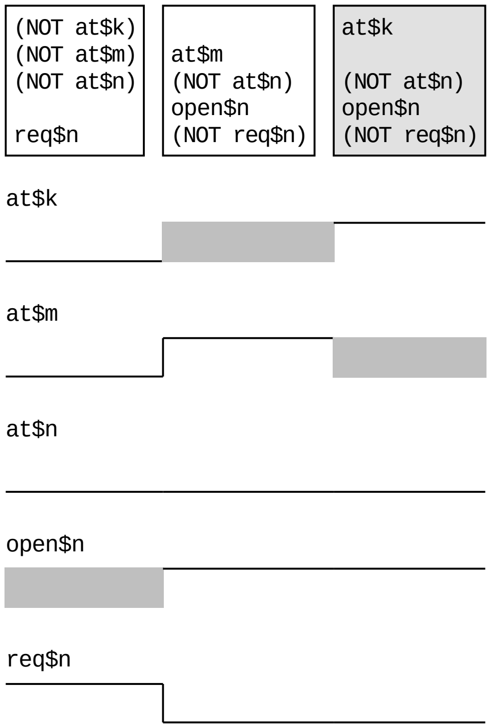


Figure 8: Countermodel generated if Safe$n is omitted in the proof of Arrive$n shown in Figure 5.

For example, to verify that Safe$1 follows from the specifications for the elevator system, the designer would first create the requirement or, if Safe$1 was created and saved in an earlier GILED session, load it from a file. The designer would then invoke `Check Proof` to begin construction of a proof or, if Safe$1 was verified previously, to determine if the proof is up-to-date and learn the premises used in the proof. If a current proof exists, the designer can opt to see the proof (i.e. the implication automatically constructed by GILED to validate the inference). If a new proof is to be attempted, GILED prompts the designer for the premises to use in the proof. In this case, the designer would designate Init, SafeOpen$1 and SafeLeave$1 as premises; GILED would then construct a proof similar to that shown in Figure 3 and check that it is valid. The designer can also prove Safe$1 directly, without invoking `Check Proof`, by building an implication representing the inference and invoking `Prove` to determine if the implication is valid.

When an attempt to verify a requirement fails, the designer can request to see a counterexample. Consider, for example, the proof of Arrive$n shown in Figure 5. If the designer overlooks the premise

Safe\$$n$ and attempts to prove that Arrive\$$n$ follows from the other four premises, GILED generates the counterexample shown in Figure 8. The model consists of an infinite sequence of states with the state predicates having the values shown in the rectangles and the shaded state infinitely repeated. (The absence of a state predicate indicates that the predicate could have any value.) GILED displays timing diagrams beneath the state sequence to aid visualization. The designer can also invoke `Construct Models` to directly generate a model that satisfies the formula in the canvas.

6 Implementation of the Toolset

Figure 9 shows the organization of the GIL tools. Rounded rectangles depict tool components (functions) and square rectangles depict data structures manipulated by the tools. A designer interacts with the tools through the mouse-driven interface provided by GILED. As described above, GILED helps the designer create new graphical formulas and retrieve and modify existing ones. It stores formulas in Unix files as abstract syntax trees with sufficient representational information to recreate the layout specified by the designer when creating them. GILED also provides the interface to the proof checker and model generator, both of which make use of an intermediate representation of a formula as a semantic tableau. The procedure that constructs the tableau requires leaner abstract syntax trees in which productions reflect the semantics, and not merely representational variations in formula. Both the proof checker and model generator communicate results back to GILED, which displays them to the designer. The tools run under the X-window system and are written in Common Lisp using the Garnet graphics toolkit [24].

A syntax directed editor for a visual language is based on an attribute grammar that specifies how a picture is represented by an annotated abstract syntax tree in which attributes provide information relating to layout. The attribute grammar used in the implementation of GILED resembles a picture layout grammar [9] but makes use of both inherited and synthesized attributes. It is similar in this regard to the grammar specifications in [5]. Each node in the abstract syntax tree representation of a GIL formula corresponds to a subpicture, which fits within a rectangular box. This is illustrated in Figure 10 by placing rectangular boxes around the main syntactic units of a sample GIL formula. Attributes associated with nodes in the abstract syntax tree specify the dimensions and co-ordinates of the corresponding boxes (subpictures). The attributes must satisfy certain constraints for a picture to represent a legal GIL formula. For example, constraints require that subformulas in a conjunction start at the same point and that an invariant is indented below

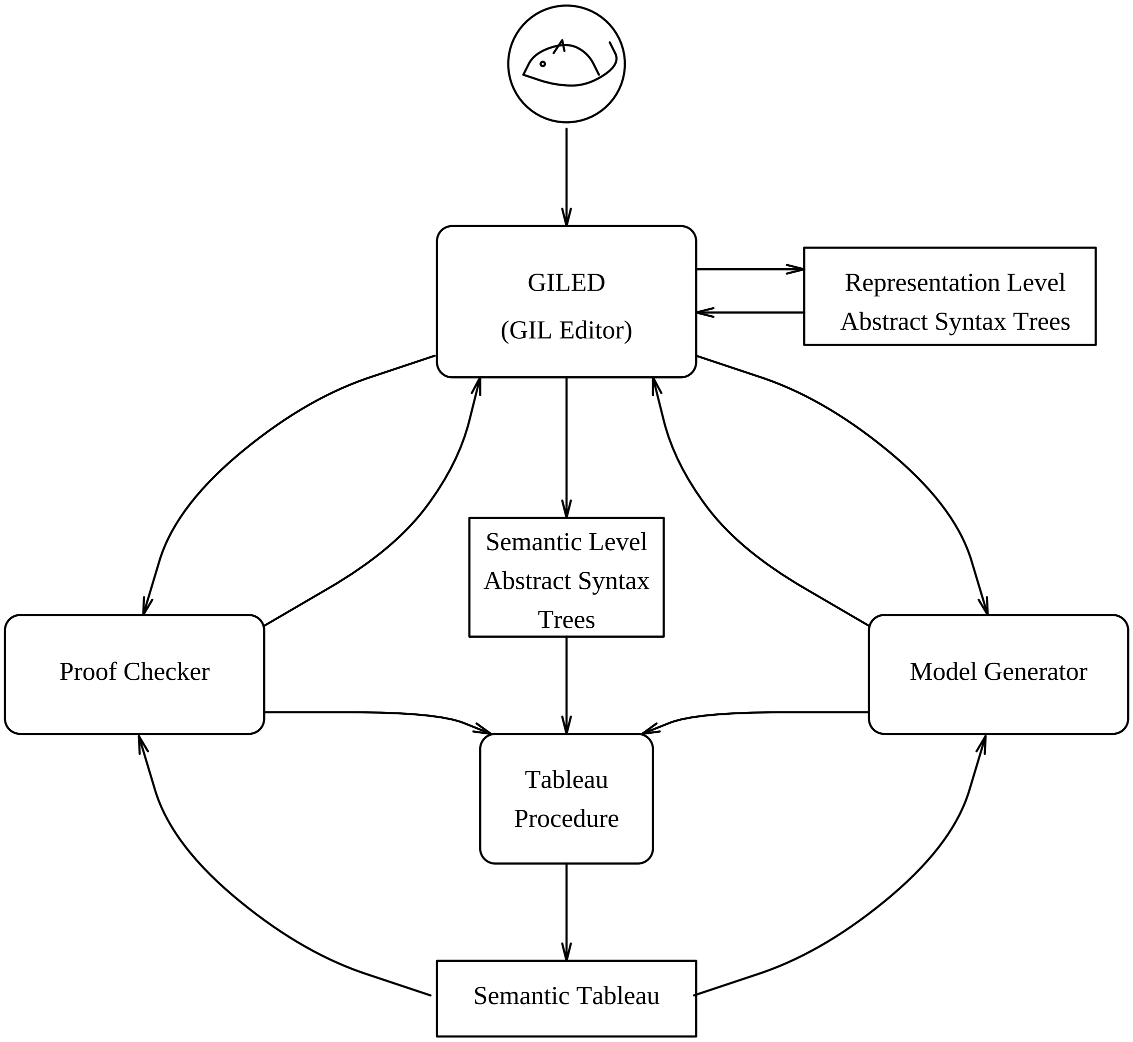


Figure 9: The GIL Tools.

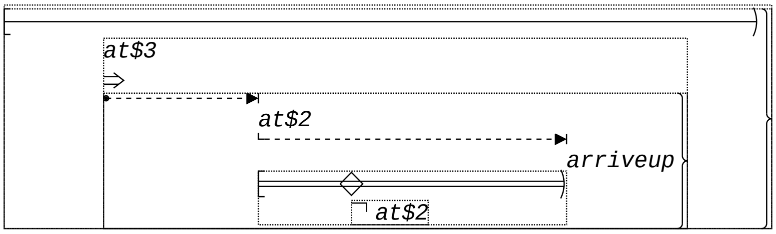


Figure 10: Structure of a sample GIL formula.

the interval it modifies.

Attribute relationships are specified by defining certain attributes to be functions of other attributes in the abstract syntax tree. Whenever the value of an attribute changes (for instance, as the result of an editing operation) values of other attributes may need to be re-computed. Our implementation makes use of the constraint maintenance system in Garnet for maintaining relationships between attributes. When GILED adds a node to an abstract syntax tree, it initializes the attributes of the node with constraints that describe attribute dependencies. Garnet maintains constraints lazily, re-computing the value of an attribute only when the value is required for the evaluation of other attributes and its current value is stale. This method results in a reasonably efficient implementation. A detailed description of the design of GILED appears in [15].

The semantics of GIL is formally defined by translation to a textual logic with a minimal set of modal operators and a model-theoretic semantics. (See Appendix B.) The textual representation facilitates the definitions of the semantics and of the decision procedure. The tableau procedure operates on a semantic level abstract syntax tree, which is an abstract syntax representation of a textual formula. Semantic level abstract syntax trees are an internal form that the tools manipulate but that the system designer cannot inspect or directly alter. The translation of a GIL formula into a semantic level abstract syntax tree is carried out by traversing the abstract syntax tree for the graphical formula, ignoring nodes and productions that have no semantic significance and converting all derived operators into equivalent interval operators.

The GIL proof checker is based on the decision procedure for the propositional form of the textual logic. The decision procedure reduces a formula to an equivalent Büchi automaton, whose accepting runs are precisely the satisfying models of the formula, and then checks the emptiness of language of the resulting automaton. Details of the automaton construction can be found in [29], which also shows the complexity of the construction is $2^{O(n^k)}$ for a formula of size n and depth k of interval nesting. The automata-theoretic method has been refined into a more traditional tableau procedure, which has lower average-case time and space requirements. To determine if a formula is valid, the proof checker negates the formula and applies the tableau construction to produce a semantic tableau that encodes the Büchi automaton for the negated formula. It then checks for accepting runs. The original formula is valid precisely if the language of the automaton is empty.

The model generator is invoked after the language of the automaton is determined to be nonempty. The model generator extracts an accepting run from the automaton; the extracted run is small in the sense that the the sum of the lengths of the initial section and the infinitely repeating section are minimized.

7 Related Work

Graphical representations of computer systems have been common in software engineering practice, but have lacked a rigorous formal basis and, thus, have tended to be illustrative and documentary rather than an integral part of the software development process. Some notable exceptions include the statechart visual formalism of Harel [12], a pictorial version of Milner's CCS, called IDCCS [7], and the $\forall$-automata of Manna and Pnueli [22]. Environments supporting the specification and verification of concurrent systems have been built around both Statecharts [13] and IDCCS. These languages are oriented toward the depiction of states and state transitions, whereas GIL focuses on showing the evolution of properties in time.

Timing Diagrams [30] is a graphical notation for expressing precedence and causality relationships between events in a computation. Like GIL, Timing Diagrams can be created using a graphical editor and checked for validity. The semantics of Timing Diagrams are defined by translation to a subset of temporal logic that can be decided very efficiently.

Allen's logic for expressing temporal relationships between intervals of time is the foundation for the TIMELOGIC temporal reasoning system [14]. The logic is textual, but graphical representations are used to show relationships among intervals more clearly.

Moszkowski's Interval Temporal Logic [10] provides an interval-like "chop" operator $\mathcal{C}$. Informally, $f\mathcal{C}g$ is true of a context if there exists a point that partitions the context into a prefix (subcontext) satisfying f and a suffix (subcontext) satisfying g. While the intuitive semantics of chop are appealing, the decision problem for formulas with chop is non-elementary in the depth of nested alternations of chop and negation. In contrast, intervals in GIL have a more operational semantics, but do not increase the complexity of the decision problem as severely. GIL can express a stronger version of chop, which suffices for expressing the properties of interest for the systems we have considered.

GIL is closest to the Interval Logic (IL) of [31], from which it is largely inspired. However, there are several presentational and semantic differences between the two logics, which we discuss briefly below.

Both IL and GIL provide explicit construction of intervals using search operations. However, they differ in the way that they construct intervals by the composition of searches. In IL, every search restricts a context and intervals are obtained by nesting searches, yielding increasingly narrower contexts. In GIL, the start and end of an interval are located independently by means of a sequential composition of searches. Searches in IL are to intervals, rather than to states at which formulas hold, as in GIL. There is no loss or gain in expressiveness in either approach, but the state-based semantics of GIL are easier to define and understand. Moreover, searching to intervals requires the introduction of event intervals, representing positive transitions of formulas, and of "begin" and "end" operators, which are used to indicate how intervals located by searching further restrict a context. IL permits searches into the past as well as the future. Allowing unrestricted searches into the past makes the decision procedure for GIL non-elementary [27]. This is a major difference from IL, where the presence of both future and past searches does not affect the complexity of the decision procedure.

Plaisted [25] demonstrated a decision procedure for IL, obtained through translation to an ω-regular expression-like language with a non-elementary decision problem. PSPACE-completeness of IL was later established by Aaby and Narayana in [1], where they give a translation of IL to an elementary fragment of a non-elementary logic. The reduction is tedious and unnatural, and points out the need for a simpler semantics that retains the advantages of being able to reason within intervals.

An experiment with a graphical representation of an IL specification for the alternating bit protocol [23] demonstrated that a visual representation results in more intuitive and natural specifications. The leaner semantics of our logic make it more amenable than IL to a clean graphical representation.

8 Conclusion

This paper has described a visual logic for specifying concurrent software systems that aids formal reasoning about temporal properties of systems. Experiments with the logic have produced graphical specifications for the sliding window protocol [16], a readers/writers database system [4], a protocol to commit transactions on a shared database [17], and a fair mutual exclusion algorithm [3], in addition to the elevator system. A prototype toolset supporting the analysis of GIL specifications has been developed.

Current research is addressing issues relating to the display of GIL formulas and the specification of temporal properties. In particular, we are experimenting with vertical spacing and scaling the size of operator symbols to improve the visual appearance of complex formulas and make their structure more visually evident. We are also investigating issues relating to the alignment of formulas to reflect known constraints on the partial ordering of points. A recent extension to GILED allows the designer to specify constraints on the ordering of points within a specification. Heuristics for recognizing search patterns that commonly occur in specifications and that impose an ordering on points in the specifications are being investigated. In such cases, GILED could aid the designer by aligning points accordingly. Methods for using a counterexample to realign the points of graphical formulas that constitute an invalid proof are also being explored. This would assist the designer in revising and correcting the proof and specifications.

GIL is very general, and certainly admits formulas that lack the immediate visualization of the sample specifications presented in Section 3. For example, the semantics of searching to a formula with nested intervals is subtle and difficult to visualize, even when the search is represented graphically. However, we have not found a need for such searches in the specifications of the concurrent systems that we have considered. Our experiments indicate that most temporal properties of interest for concurrent systems can be specified in a natural and visually appealing manner using the derived operators introduced in this paper. On-going research is attempting to identify syntactic restrictions that permit inferences to be checked more efficiently and still allow natural specifications of concurrent systems.

A real-time extension of GIL [28] provides primitives for bounding the duration of intervals. We have recently modified the GIL proof checker to validate deductions in the extended logic and are currently experimenting with its use. We are also investigating the integration of the GIL decision procedure with an automated reasoning system that provides decision procedures for other useful theories, such as linear inequalities and Presburger arithmetic, and that provides better support for the management of proofs.

The GIL toolset is a prototype. It was developed to demonstrate proof-of-concept and to facilitate experiments with the logic and its graphical representation. Both the logic and the display of formulas have evolved based on our experience with the tools. We expect this process of experimentation and revision to continue as we refine the current toolset into a working environment for specification, validation and design of concurrent software systems. A robust, user-friendly environment will permit empirical studies needed to determine whether software designers find a visual logic, such as GIL, easier to use than a textual logic.

References

[1] A. A. Aaby and K. T. Narayana. Propositional temporal interval logic is PSPACE complete. In *Proc. 9th Inter. Conf. Automated Deduction*, pp. 218–237, Argonne IL, May 1988. LNCS 193, Springer-Verlag.

[2] H. Barringer, R. Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. In *Proc. 16th ACM Symp. Theory of Computing*, pp. 51–63, Washington, D.C., Apr. 1984.

[3] L. K. Dillon, G. Kutty, P. M. Melliar-Smith, L. E. Moser, and Y. S. Ramakrishna. Visual specifications for temporal reasoning. Submitted, May 1993.

[4] L. K. Dillon, G. Kutty, L. E. Moser, P. M. Melliar-Smith, and Y. S. Ramakrishna. Graphical specifications for concurrent software systems. In *Proc. 14th IEEE Inter. Conf. Software Engineering*, pp. 213–224, Melbourne, May 1992.

[5] P. Franchi-Zannettacci. Attribute specifications for graphical interface specifications. In G. X. Ritter, ed., *Information Processing '89*, pp. 149–155. IFIP, Elsevier Science Publishers B.V., North Holland, 1989.

[6] D. M. Gabbay. The declarative past and imperative future. In *Proc. Colloq. Temporal Logic in Specification*, pp. 409–448, 1987. LNCS 398, Springer-Verlag.

[7] A. Giacalone and S. A. Smolka. Integrated environments for formally well-founded design and simulation of concurent systems. *IEEE Trans. Software Engineering*, 14(6):787–801, June 1988.

[8] W. D. Gillett and T. D. Kimura. Parsing two-dimensional languages. In *Proc. IEEE 10th Inter. Computer Software and Applications Conf.*, pp. 472–477, Chicago, Oct. 1986.

[9] E. Golin and S. P. Reiss. The specification of visual language syntax. In *Proc. IEEE Work. Visual Languages*, pp. 105–110, Rome, Oct. 1989.

[10] J. Y. Halpern, Z. Manna, and B. Moszkowski. A hardware semantics based on temporal intervals. In *Proc. 10th Inter. Conf. Automata, Languages and Programming*, pp. 278–291, Barcelona, 1983.

[11] J. Y. Halpern and Y. Shoham. A propositional modal logic of time intervals. *J. ACM*, 38(4):935–962, Oct. 1991.

[12] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[13] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Trans. Software Engineering*, 16(4):403–414, Apr. 1990.

[14] J. A. G. M. Koomen. The timelogic temporal reasoning system. Tech. Rep., Dept. Computer Science, University of Rochester, NY, Nov. 1987. (Revised March 1989).

[15] G. Kutty. A tool for the interactive generation of Graphical Interval Logic formulas. Tech. Rep. 9307, Dept. Electrical and Computer Engineering, University of California, Santa Barbara, Mar. 1993.

[16] G. Kutty, Y. S. Ramakrishna, L. K. Dillon, L. E. Moser, and P. M. Melliar-Smith. Specification of a communication protocol in graphical interval logic. In *Proc. IEE Inter. Conf. Information Engineering*, pp. 432–441, Singapore, Dec. 1991.

[17] G. Kutty, Y. S. Ramakrishna, L. E. Moser, L. K. Dillon, and P. M. Melliar-Smith. A graphical interval logic tooset for verifying concurrent systems. In *Proc. 4th Conf. Computer Aided Verification*, pp. 138–153, Elounda, Greece, July 1993. LNCS 697, Springer-Verlag.

[18] L. Lamport. What good is temporal logic? In *Proc. IFIP Congress*, pp. 657–668, Paris, 1983.

[19] L. Lamport. A temporal logic of actions. Tech. Rep. 57, DEC Systems Research Center, Palo Alto, CA, Apr. 1990.

[20] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, Mar. 1966.

[21] Z. Manna and A. Pnueli. Verification of concurrent programs: The temporal framework. In R. S. Boyer and J. S. Moore, eds., *The Correctness Problem in Computer Science*, pp. 215–273. Academic Press, 1981.

[22] Z. Manna and A. Pnueli. Specification and verification of concurrent programs by ∀-automata. In *Proc. Conf. Temporal Logic in Specification*, pp. 124–187, Altrincham, England, Apr. 1987. LNCS 398, Springer-Verlag.

[23] P. M. Melliar-Smith. A graphical representation of interval logic. In *Proc. Inter. Conf. Concurrency*, pp. 106–120, Hamburg, FRG, Oct. 1988. LNCS 335, Springer-Verlag.

[24] B. A. Myers, D. A. Guise, R. B. Dannenberg, B. Vander Zanden, D. S. Kosbie, E. Pervin, A. Mickish, and P. Marchal. Garnet: Comprehensive support for graphical highly interactive user interfaces. *IEEE Computer*, 18(11):71–85, Nov. 1990.

[25] D. Plaisted. A low level language for obtaining decision procedures for classes of temporal logics. In R. Schwartz, ed., *An Interval Logic for Higher Level Temporal Reasoning*, pp. 1–35. NASA Contractor Report 172262, Sept. 1983.

[26] V. Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.

[27] Y. S. Ramakrishna. *Interval Logics for Temporal Specification and Verification.* PhD thesis, Dept. Computer and Electrical Engineering, University of California, Santa Barbara. In preparation.

[28] Y. S. Ramakrishna, L. K. Dillon, L. E. Moser, P. M. Melliar-Smith, and G. Kutty. A real-time interval logic and its decision procedure. Submitted.

[29] Y. S. Ramakrishna, L. K. Dillon, L. E. Moser, P. M. Melliar-Smith, and G. Kutty. An automata-theoretic decision procedure for future interval logic. In *Proc. 12th Conf. Foundations of Software Technology and Theoretical Computer Science*, pp. 51–67, New Delhi, Dec. 1992. LNCS 652, Springer-Verlag.

[30] R. Schlör and W. Damm. Specification of system-level hardware designs using timing diagrams. In *Proc. Europ. Conf. Design Automation and Europ. Event in ASIC Design*, pp. 518–524, Paris, Feb. 1993. IEEE Computer Society Press.

[31] R. L. Schwartz, P. M. Melliar-Smith, and F. H. Vogt. An interval logic for higher-level temporal reasoning. In *Proc. 2nd ACM Symp. Principles of Distributed Computing*, pp. 173–186, Montreal, Canada, Aug. 1983.

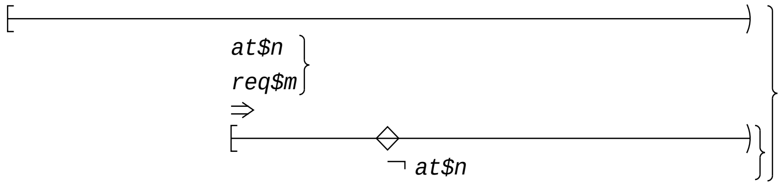
[32] P. Wolper. On the relation of programs and computations to models of temporal logic. In *Proc. Conf. Temporal Logic in Specification*, pp. 75–123, Altrincham, England, Apr. 1987. LNCS 398, Springer-Verlag.

A Proof Structure

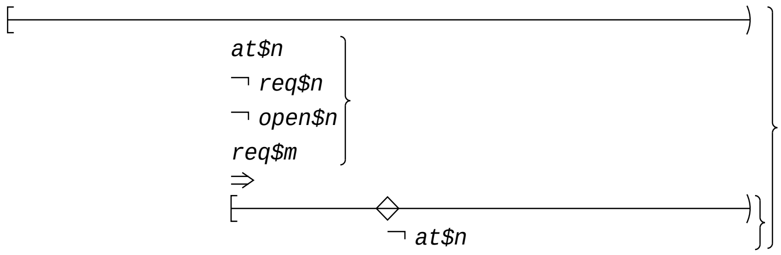
The intermediate lemmas used in the proofs of Arrive\$n-At\$m, $n, m = 1, 2, 3$ and $n \neq m$, are listed below. Also given are proof trees for the lemmas and for the arrival requirements Arrive\$n-At\$m.

A.1 Departure Lemmas $(n, m = 1, 2, 3,\ n \neq m)$

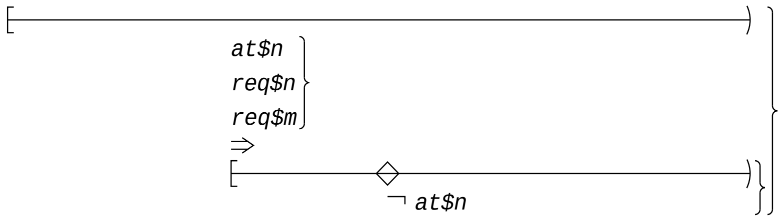
Depart\$$n$-Req\$$m$:



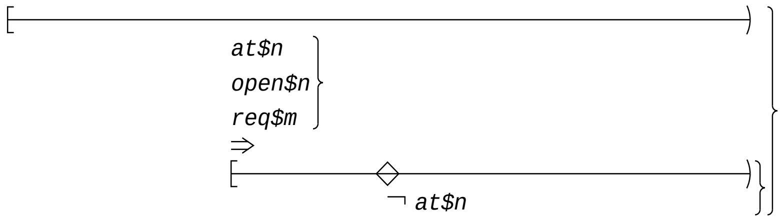
Depart\$$n$-Req\$$m$-NOpen-NReq:



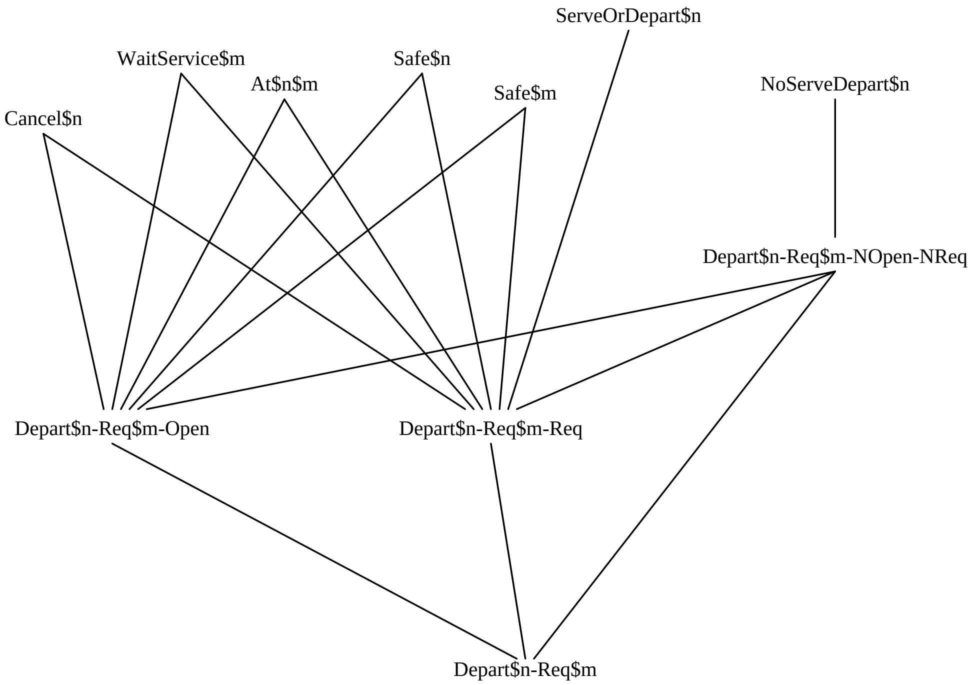
Depart\$$n$-Req\$$m$-Req:



Depart\$$n$-Req\$$m$-Open:

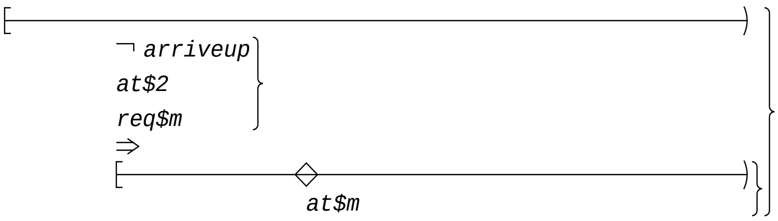


A.2 Proof Tree for Departure Lemmas

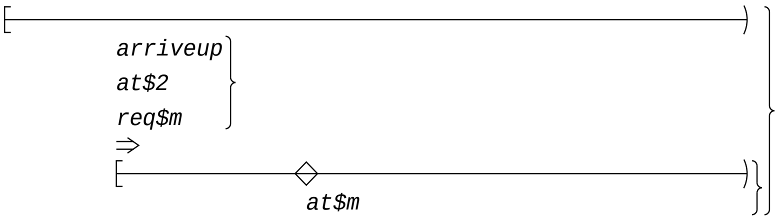


A.3 Arrival Lemmas $(m = 1, 3)$

Arrive\$$m$-At\$2-Down:

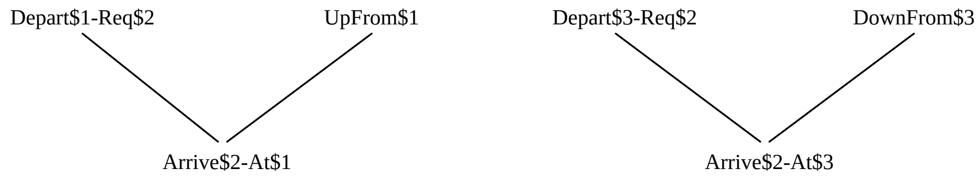


Arrive\$$m$-At\$2-Up:

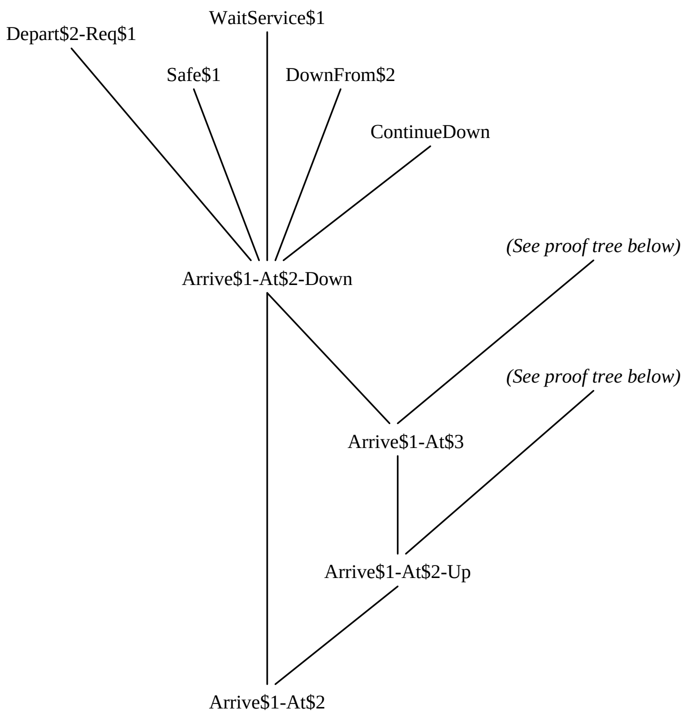


A.4 Proof Trees for Arrival Lemmas

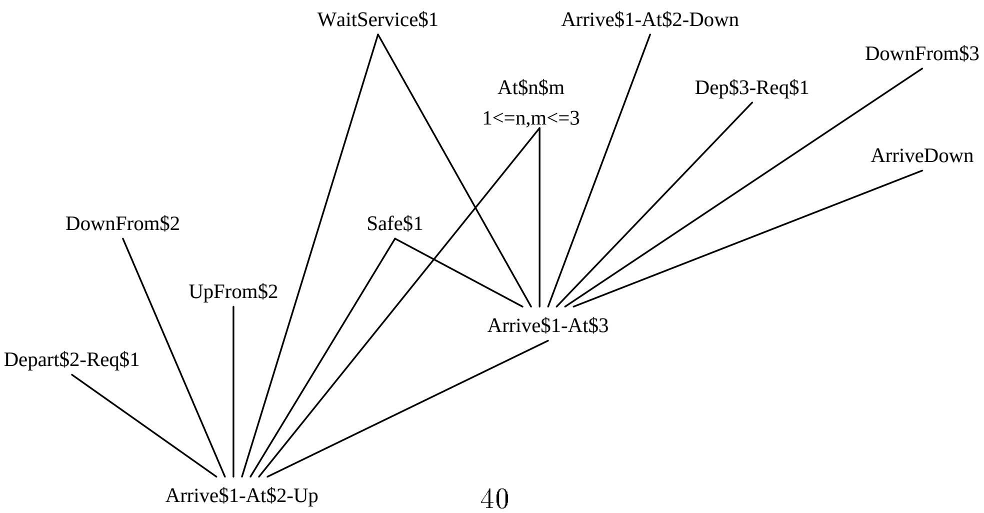
Proof trees for Arrive\$2-At\$1 and Arrive\$2-At\$3:



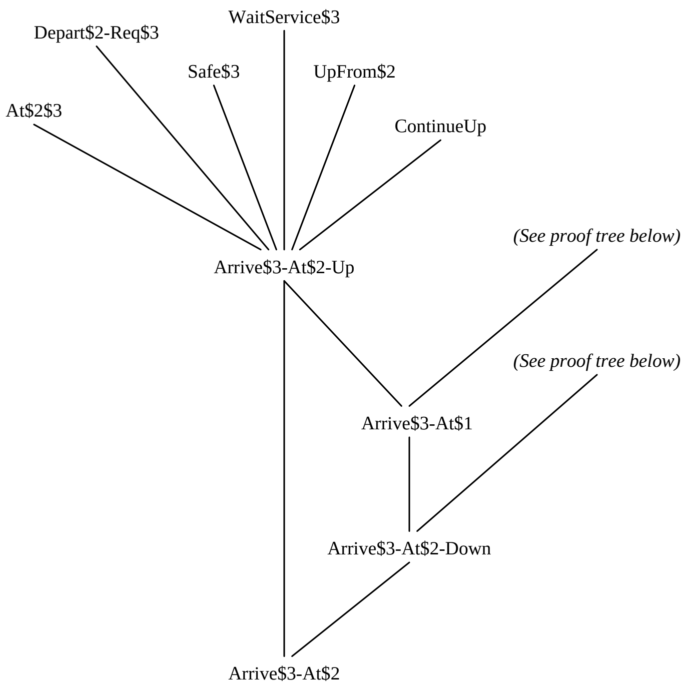
Proof tree for Arrive\$1-At\$2-Down and Arrive\$1-At\$2:



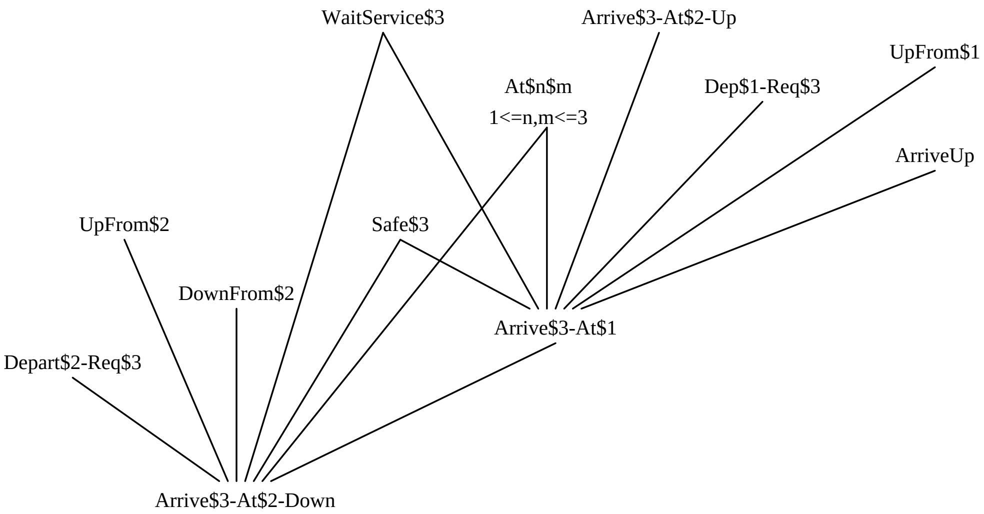
Proof tree for Arrive\$1-At\$3 and Arrive\$1-At\$2-Up:



Proof tree for Arrive$3-At$2-Up and Arrive$3-At$2:



Proof tree for Arrive$3-At$1 and Arrive$3-At$2-Down:



B Semantics

A model-theoretic semantics for our interval logic is presented below. For convenience, the semantics make use of a linear version of the logic, called Future Interval Logic[4] (FIL). The semantics of GIL

[4] "Future" because formulas assert properties of a state's future context.

is then obtained by translation from GIL to FIL.

B.1 Syntax of FIL

The language of FIL, like that of GIL, is defined relative to a set $\mathcal{P}$ of state predicates. The definition below makes use of the following generic symbols.

State Predicates: p, p_1, p_2, ...

Primitive search patterns: q, q_1, q_2, ...

General search patterns: Q, Q_1, Q_2, ...

Intervals: I, I_1, I_2, ...

Formulas: F, F_1, F_2, ..., G

The linear syntax of FIL is defined as follows.

$$F ::= p \;\|\; F_1 \vee F_2 \;\|\; \neg F_1 \;\|\; (F_1) \;|\; I F_1$$
$$I ::= [Q_1|Q_2)$$
$$Q ::= q \;\|\; q, Q_1$$
$$q ::= \rightarrow F \;\|\; \rightarrow$$

For convenience, we extend FIL with several abbreviations. As usual, $F_1 \wedge F_2 = \neg(\neg F_1 \vee \neg F_2)$. The temporal Henceforth and Eventually operations are also defined: $\Box F = [\rightarrow \neg F|\rightarrow)\mathit{false}$ and $\Diamond F = \neg[\rightarrow F|\rightarrow)\mathit{false}$. The shorthand $-$ denotes the trivial search $\rightarrow \mathit{true}$. The brackets $[\![$ and $)\!)$ signify a strong interval, which is defined

$$[\![Q_1|Q_2)\!)F = [Q_1|Q_2)F \wedge ([Q_1|\rightarrow)\mathit{false} \vee [Q_2|\rightarrow)\mathit{false} \vee \neg[Q_1|Q_2)\mathit{false}).$$

Finally, we define a strong search to a target F, denoted $\rightarrow * F$, as follows:

$$\{Q_1, \rightarrow * F, Q_2|Q_3\}G \quad = \quad \{Q_1, \rightarrow F, Q_2|Q_3\}G \wedge [Q_1|\rightarrow)\Diamond F$$
$$\{Q_1|Q_2, \rightarrow * F, Q_3\}G \quad = \quad \{Q_1|Q_2, \rightarrow F, Q_3\}G \wedge ([Q_1|\rightarrow)\mathit{false} \vee [Q_2|\rightarrow)\Diamond F$$

where the brackets { and } may be replaced by either pair of interval bracket delimiters: [and) or $[\![$ and $)\!)$. The rules that define strong searches and strong intervals can be shown to be semantically confluent, so that strong searches and strong intervals can be expanded in any order.

B.2 Semantics of FIL

A *model* (s, i) for an FIL formula consists of a *context* s and an *index* i. A context is either an infinite sequence of states $s = s(0), s(1), \ldots$ or the *null* context $\perp$. A *state* $s(i)$ in a non-null context s assigns valuations to the state predicates in $\mathcal{P}$. We identify a state $s(i) \subseteq \mathcal{P}$ with the collection of state predicates true in that state. The index i in a model is a (finite) nonnegative integer. We denote by $F|_{s,i}$ the value of a formula F in the model (s, i).

For the definitions below, we extend the set of nonnegative integers with an infinite element ω, satisfying $i < \omega$ for all finite i, and define addition and subtraction on nonnegative integers (including ω) in the usual manner. For a non-null state sequence s and indices l and r satisfying $l \leq r$ and $r < \omega$, we further define a sequence $s_{\langle l,r \rangle}$ as follows.

$$s_{\langle l,r \rangle}(k) = \begin{cases} s(k+l) & \text{for } 0 \leq k < r - l + 1 \\ s(r) & \text{for } r - l + 1 \leq k < \omega \end{cases}$$

Thus, $s_{\langle l,r \rangle}$ denotes the context obtained from s by extracting the subsequence from state $s(l)$ through state $s(r)$ and stuttering the final state.

The following rules provide an inductive definition for the truth value of an FIL formula. They make use of the function Construct for constructing the subcontext specified by an interval. Assuming $s \neq \perp$, we define

- $F|_{\perp,i} = \mathit{true}$.
- $p|_{s,i} = \mathit{true}$ if and only if $p \in s(i)$.
- $(F_1 \vee F_2)|_{s,i} = \mathit{true}$ if and only if $F_1|_{s,i} = \mathit{true}$ or $F_2|_{s,i} = \mathit{true}$.
- $(\neg F_1)|_{s,i} = \mathit{true}$ if and only if $F_1|_{s,i} = \mathit{false}$.
- $(F_1)|_{s,i} = \mathit{true}$ if and only if $F_1|_{s,i} = \mathit{true}$.
- $I F_1|_{s,i} = \mathit{true}$ if and only if $F_1|_{s',0} = \mathit{true}$,
  where $s' = \text{Construct}(s, i, I)$.

The definition of Construct makes use of the function Locate for locating the state specified by a search pattern. Locate is defined as follows.

- $\text{Locate}(\rightarrow, s, i) = \omega$

- $\text{Locate}(\rightarrow F, s, i) = \begin{cases} \min(K) & \text{if } K \neq \emptyset \\ \epsilon & \text{otherwise} \end{cases}$

  where $K = \{\, k \mid \ \omega > k \geq i \text{ and } F|_{s,k} = \mathit{true} \,\}$ and the special error value ϵ signifies a failed search

- $\text{Locate}((q, Q), s, i) = \begin{cases} \text{Locate}(Q, s, \text{Locate}(q, s, i)) & \text{if Locate}(q, s, i) \neq \epsilon \\ \epsilon & \text{otherwise} \end{cases}$

Given a context, an index, and an interval, Construct produces the subcontext represented by the interval:

- $\text{Construct}(s, i, [Q_1|Q_2)) = \begin{cases} \perp & \text{if } r < l,\, l = \epsilon \text{ or } r = \epsilon \\ s_{\langle l,r \rangle} & \text{otherwise} \end{cases}$

  where $l = \text{Locate}(Q_1, s, i)$, $r = \text{Locate}(Q_2, s, i) - 1$, and $\epsilon - 1 = \epsilon$.

B.3 Formal Syntax for GIL and Translation Rules

The syntax of GIL is specified using a generalization of the attributed multiset grammar model described in [9]. A multiset grammar differs from a context-free grammar in that its productions rules do not impose any order on the symbols in their right-hand sides. A multiset grammar defines a set of multisets (unordered collections of terminal symbols, possibly containing repeated elements) rather than a set of strings. An attributed multiset grammar augments a multiset grammar with

- a set of attributes, which play an integral role in parsing an input
- semantic functions, which define the values of the attributes, and
- constraints, which indicate when a production can be applied.

In the attributed multiset grammar model of [9], parsing attributes are restricted to synthesized attributes. For defining the syntax of GIL, however, we require a more general grammar model, which permits both synthesized and inherited attributes to be used for parsing. The grammar given below can be viewed as belonging to the index set grammar model described in [8].

In the sequel, therefore, an attributed multiset grammar consists of

- a finite set of terminal symbols
- a finite set of nonterminal symbols

- a finite set of attributes
- a mapping that associates sets of attributes with the terminal and nonterminal symbols and
- a finite set of productions.

Attributes are classified as synthesized or inherited, and attributes associated with terminal symbols are restricted to synthesized attributes. A production specifies a rewrite rule, which can be used to expand the nonterminal on the LHS into the multiset of symbols on the RHS, and associates a finite set of semantic functions and a finite set of constraints with the rule. Each semantic function defines an attribute of one of the rule's nonterminal symbols as a function of the values of other attributes of symbols in the rule. A production provides semantic functions for each synthesized attribute of the nonterminal on a rule's LHS and for each inherited attribute of the nonterminals on the rule's RHS. As is customary, the rewrite rules and semantic functions in a grammar must not admit derivations in which attribute values are defined circularly. The constraints associated with a rule are defined over inherited attributes of the nonterminal on the rule's LHS and synthesized attributes of symbols on the rule's RHS. The constraints specify conditions that must be satisfied in order to apply the rule.

The terminal symbols in our grammar for GIL correspond to the GIL operators, search arrows, interval symbols and brackets, which are described in Section 2, and to state formulas, which we denote by the special terminal symbol *state-form*.[5] The nonterminals are defined as follows.

- F, representing GIL formulas
- Qq, representing a pair of search patterns
- Q, representing a single search pattern
- Qv, representing the continuation of a search pattern[6]
- L, representing a line segment denoting an interval
- Ai, representing a search arrow that begins a search pattern and
- Av, representing a search arrow that is embedded in a search pattern.

[5] State formulas are parsed using a context free grammar. We omit the details, which are standard.
[6] For simplicity, we do not permit the horizontal shorthand for composing searches used in Section 2.

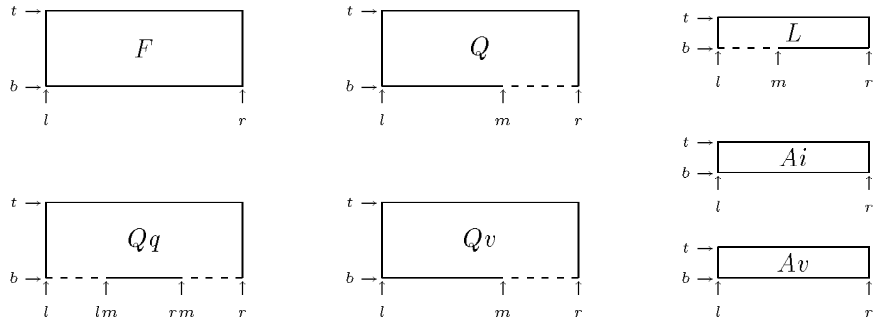


Figure 11: Nonterminal symbols and their attributes.

We regard each instance of a terminal or nonterminal symbol as enclosed by a bounding box. Synthesized attributes associated with the symbols give the position and dimensions of this box:

- t gives the y-coordinate of the top of the box.
- b gives the y-coordinate of the bottom of the box.
- l gives the x-coordinate of the left side.
- r gives the x-coordinate of the right side.

An additional synthesized attribute is associated with interval symbols that denote eventualities:

- p gives the x-coordinate of the center of the diamond.

The remaining attributes are inherited. Two inherited attributes are associated with the nonterminal Qq representing a pair of search patterns:

- lm gives the x-coordinate of the point that the first search pattern locates.
- rm gives the x-coordinate of the point that the second search pattern locates

Finally, the grammar associates an inherited attribute with each of the nonterminals Q, Qv, and L:

- m gives the x-coordinate of the point that a search pattern Q or Qv locates or gives the x-coordinate of the formula that modifies an interval L.

Figure 11 illustrates the relationships between the attributes for nonterminal symbols. It also illustrates the conventions we use when drawing nonterminals. Lines show order relations between attribute values (vertical lines for y-coordinates and horizontal lines for x-coordinates), with strict ordering denoted by solid lines and nonstrict ordering denoted by broken lines. Thus, for example, the attributes associated with Qq are subject to the following constraints: $b < t$, $l \leq lm$, $lm < rm$ and $rm \leq r$.

Productions are shown in Tables 1–6. The last column of the tables also defines the translation of GIL to FIL. Table 1 gives sample productions and a translation scheme for the root nonterminal F.[7] Each row in Table 1 represents a production for F and the corresponding translation rule. The second column shows the RHS of the rewrite rules, which are named, in the first column, for reference purposes below. Rewrite rules are shown graphically using a two-dimensional format so that relationships among attribute values, which are formally expressed by the semantic functions and constraints, are visually apparent. We use a broken line for the right side of a box when the right side need not coincide with the right side of the box enclosing the LHS nonterminal. The third and fourth columns give the constraints and semantic functions, respectively. Unsubscripted attribute names refer to attributes of the LHS nonterminal and subscripted attribute names refer to attributes of the symbols on the RHS. The fifth column of Table 1 defines the translation of F into a string $t(F)$ representing an FIL formula. For simplicity, we give a fully parenthesized translation.

Tables 2, 3 and 4 provide productions and translation rules for Qq, Q, and Qv, respectively. As shown, Q and Qv differ only in the type of arrow with which an instance begins. Table 5 defines three translation functions for line segments: t_{lt} generates a left interval-delimiter, t_{rt} generates a right interval-delimiter, and t_{md} generates a Henceforth or Eventually symbol or, when the formula that modifies the interval is positioned at the first state of the interval, an empty string. Table 6 gives the productions and translations of both types of search arrows. The rewrite rules for the two kinds of arrows do not require any constraints, and their semantic functions and translations are identical.

Figure 12 shows a parse tree for the formula ArriveDown in the specification of the elevator system. Attribute values are shown in the table below the parse tree. The top part of the table represents the input (terminal symbols). We show some steps in the translation of ArriveDown

[7] For brevity, we have omitted productions for some of the propositional operators. Productions and translations for the missing operators are similar to those given for implication.

Rule	F	Constraints	Sem. Functions	$t(F)$
F1	$\neg_1$ F_2	$l_1 = l_2,\ t_2 \leq b_1$	$t \leftarrow t_1;\ b \leftarrow b_2;$ $l \leftarrow l_1;\ r \leftarrow r_2$	“$\neg$(” $t(F_2)$ “)”
F2	F_1 F_2	$l_1 = l_2,\ t_2 \leq b_1$	$t \leftarrow t_1;\ b \leftarrow b_2;$ $l \leftarrow l_1;$ $r \leftarrow \max\{\, r_i \,\}_i$	“(” $t(F_1)$ “) $\wedge$ (” $t(F_2)$ “)”
F3	F_1 $\Rightarrow_2$ F_3	$l_1 = l_2 = l_3,$ $t_2 \leq b_1,\ t_3 \leq b_2$	$t \leftarrow t_1;\ b \leftarrow b_3;$ $l \leftarrow l_1;$ $r \leftarrow \max\{\, r_i \,\}_i$	“(” $t(F_1)$ “) $\Rightarrow$ (” $t(F_2)$ “)”
F4	F_2 $\}_1$	$t_2 \leq t_1,\ b_1 \leq b_2,$ $r_2 \leq l_1$	$t \leftarrow t_1;\ b \leftarrow b_1;$ $l \leftarrow l_2;\ r \leftarrow r_1$	“(” $t(F_2)$ “)”
F5	Qq_2 L_3 F_4 $\}_1$	$t_2 \leq t_1,\ t_3 \leq b_2,$ $t_4 \leq b_3,\ b_1 \leq b_4,$ $l_2 \leq l_3,\ l_3 \leq l_4,$ $r_3 \leq r_2,\ r_4 \leq r_3,$ $r_2 \leq l_1$	$t \leftarrow t_1;\ b \leftarrow b_1;$ $l \leftarrow l_2;\ r \leftarrow r_1;$ $lm_2 \leftarrow l_3;$ $m_3 \leftarrow l_4;$ $rm_2 \leftarrow r_3$	$t_{lt}(L_3)t(Qq_2)t_{rt}(L_3)$ $t_{md}(L_3)$“(” $t(F_4)$ “)”
F6	L_2 F_3 $\}_1$	$t_2 \leq t_1,\ t_3 \leq b_2,$ $b_1 \leq b_3,\ l_2 \leq l_3,$ $r_2 \leq l_1\ r_3 \leq r_2$	$t \leftarrow t_1;\ b \leftarrow b_1;$ $l \leftarrow l_2;\ r \leftarrow r_1;$ $m_2 \leftarrow l_3$	$t_{md}(L_2)$“(” $t(F_3)$ “)”
F7	Q_2 $\triangle_3$ F_4 $\}_1$	$t_2 \leq t_1,\ t_3 \leq b_2,$ $t_4 \leq b_3,\ b_1 \leq b_4,$ $l_2 \leq l_3,\ r_3 \leq r_2,$ $l_4 = (l_3 + r_3)/2,$ $\max\{\, r_i \,\}_i \leq l_1,$	$t \leftarrow t_1;\ b \leftarrow b_1;$ $l \leftarrow l_2;\ r \leftarrow r_1;$ $m \leftarrow l_4$	“[” $t(Q_2)$ “$\vert\!\rightarrow$)(” $t(F_4)$ “)”
F8	*state-form*$_1$		$t \leftarrow t_1;\ b \leftarrow b_1;$ $l \leftarrow l_1;\ r \leftarrow r_1$	*state-form*$_1$

Table 1: Translation rules for a GIL formula F

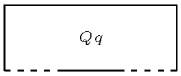
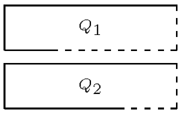
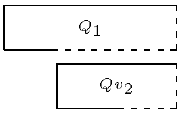
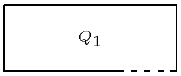
Rule		Constraints	Sem. Functions	$t(Qq)$
Qq1		$l_1 = l_2,\ t_2 \leq b_1,$ $l_1 < lm \leq r_1,$ $rm \leq r_2$	$t \leftarrow t_1;\ b \leftarrow b_2;\ l \leftarrow l_1;$ $r \leftarrow \max\{\, r_i \,\}_i;$ $m_1 \leftarrow lm;\ m_2 \leftarrow rm$	$t(Q_1)$"|"$t(Q_2)$
Qq2		$l_1 < l_2,\ t_2 \leq b_1,$ $l_2 = lm \leq r_1,$ $rm \leq r_2$	$t \leftarrow t_1;\ b \leftarrow b_2;\ l \leftarrow l_1;$ $r \leftarrow \max\{\, r_i \,\}_i;$ $m_1 \leftarrow lm;\ m_2 \leftarrow rm$	$t(Q_1)$"|" $t(Q_1)$","$t(Qv_2)$
Qq3		$l_1 = lm,$ $rm \leq r_1$	$t \leftarrow t_1;\ b \leftarrow b_1;\ l \leftarrow l_1;$ $r \leftarrow r_1;\ m_1 \leftarrow rm$	"–|"$t(Q_1)$

Table 2: Translation rules for a search pair Qq

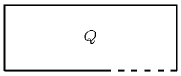
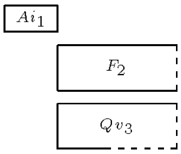
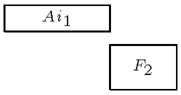
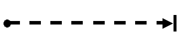
Rule		Constraints	Sem. Functions	$t(Q)$
Q1		$r_1 < m,$ $r_1 = l_2 = l_3,$ $t_2 \leq b_1,\ t_3 \leq b_2,$ $m \leq r_3$	$t \leftarrow t_1;\ b \leftarrow b_3;\ l \leftarrow l_1;$ $r \leftarrow \max\{\, r_i \,\}_i;\ m_3 \leftarrow m$	$t(Ai_1)t(F_2)$","$t(Qv_3)$
Q2		$r_1 = m,\ l_2 = r_1,$ $t_2 \leq b_1$	$t \leftarrow t_1;\ b \leftarrow b_2;\ l \leftarrow l_1;$ $r \leftarrow r_2$	$t(Ai_1)t(F_2)$
Q3		$r_1 = m$	$t \leftarrow t_1;\ b \leftarrow b_1;\ l \leftarrow l_1;$ $r \leftarrow r_1$	"$\rightarrow$"

Table 3: Translation rules for a search pattern Q (vertical layout)

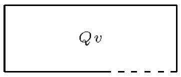
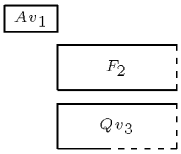
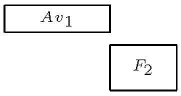
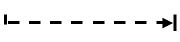
Rule		Constraints	Sem. Functions	$t(Qv)$
Qv1		$r_1 < m,$ $r_1 = l_2 = l_3,$ $t_2 \leq b_1,\ t_3 \leq b_2,$ $m \leq r_3$	$t \leftarrow t_1;\ b \leftarrow b_3;\ l \leftarrow l_1;$ $r \leftarrow \max\{\, r_i \,\}_i;\ m_3 \leftarrow m$	$t(Av_1)t(F_2)$","$t(Qv_3)$
Qv2		$r_1 = m,\ l_2 = r_1,$ $t_2 \leq b_1$	$t \leftarrow t_1;\ b \leftarrow b_2;\ l \leftarrow l_1;$ $r \leftarrow r_2$	$t(Av_1)t(F_2)$
Qv3		$r_1 = m$	$t \leftarrow t_1;\ b \leftarrow b_1;\ l \leftarrow l_1;$ $r \leftarrow r_1$	"$\rightarrow$"

Table 4: Translation rules for the continuation Qv of a search pattern (vertical layout)

Rule	L	Constraints	Sem. Functions	$t_{lt}(I)$	$t_{rt}(L)$	$t_{md}(L)$
L1			$t \leftarrow t_1;\ b \leftarrow b_1;$ $l \leftarrow l_1;\ r \leftarrow r_1$	“[”	“)”	if $l_1 < m$: “□” if $l_1 = m$: “ ”
L2			$t \leftarrow t_1;\ b \leftarrow b_1;$ $l \leftarrow l_1;\ r \leftarrow r_1$	“$\llbracket$”	“$\rrparenthesis$”	if $l_1 < m$: “□” if $l_1 = m$: “ ”
L3		$m = p_1$	$t \leftarrow t_1;\ b \leftarrow b_1;$ $l \leftarrow l_1;\ r \leftarrow r_1$	“[”	“)”	“◇”
L4		$m = p_1$	$t \leftarrow t_1;\ b \leftarrow b_1;$ $l \leftarrow l_1;\ r \leftarrow r_1$	“$\llbracket$”	“$\rrparenthesis$”	“◇”

Table 5: Translation rules for a line segment L

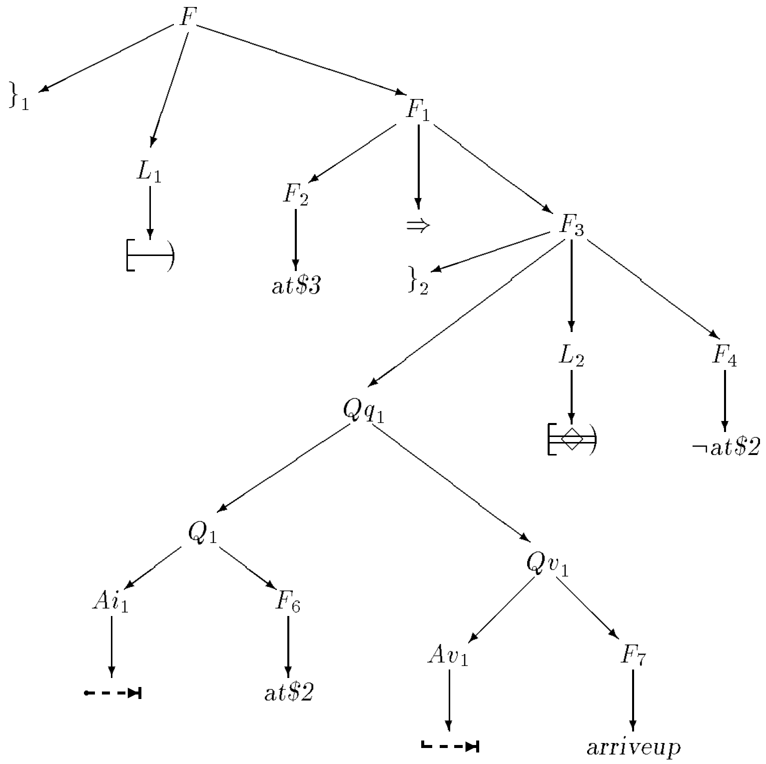
Rule	Ai	Rule	Av	Sem. Functions	$t(Ai)/t(Av)$
Ai1		Av1		$t \leftarrow t_1;\ b \leftarrow b_1;$ $l \leftarrow l_1;\ r \leftarrow r_1$	“$\rightarrow$”
Ai2		Av2		$t \leftarrow t_1;\ b \leftarrow b_1;$ $l \leftarrow l_1;\ r \leftarrow r_1$	“$\rightarrow *$”

Table 6: Translation rules for the arrows Ai and Av

below. Annotations over a derivation arrow indicate the translation rules used at each step.

$$
\begin{aligned}
t(F) &\xrightarrow{\text{F6}} t_{md}(L_1)\text{“}(\text{”}t(F_1)\text{“})\text{”} \\
&\xrightarrow{\text{L1}} \text{“}\Box(\text{”}t(F_1)\text{“})\text{”} \\
&\xrightarrow{\text{F3,F8}} \text{“}\Box((at\$3) \Rightarrow (\text{”}t(F_3)\text{“}))\text{”} \\
&\xrightarrow{\text{F5}} \text{“}\Box((at\$3) \Rightarrow (\text{”}t_{lt}(L_2)t(Qq_1)t_{rt}(L_2)t_{md}(L_2)\text{“}(\text{”}t(F_4)\text{“})))\text{”} \\
&\xrightarrow{\text{L4}} \text{“}\Box((at\$3) \Rightarrow (\llbracket\text{”}t(Qq_1)\text{“}\rrparenthesis\Diamond(\text{”}t(F_4)\text{“})))\text{”} \\
&\xrightarrow{\text{Qq2}} \text{“}\Box((at\$3) \Rightarrow (\llbracket\text{”}t(Q_1)\text{“}\,|\,\text{”}t(Q_1)\text{“},\text{”}t(Qv_1)\text{“}\rrparenthesis\Diamond(\text{”}t(F_4)\text{“})))\text{”} \\
&\xrightarrow{\text{Q2,Ai1,F8}} \text{“}\Box((at\$3) \Rightarrow (\llbracket\rightarrow at\$2|\rightarrow at\$2,\text{”}t(Qv_1)\text{“}\rrparenthesis\Diamond(\text{”}t(F_4)\text{“})))\text{”} \\
&\xrightarrow{\text{Qv2,Av1,F8}} \text{“}\Box((at\$3) \Rightarrow (\llbracket\rightarrow at\$2|\rightarrow at\$2,\rightarrow arriveup\rrparenthesis\Diamond(\neg at\$2)))\text{”}
\end{aligned}
$$

The GIL toolset does not include a parser. Formulas are constructed using a syntax-directed editor. We are currently investigating the effects of characteristics of attributed multiset grammars, such as presented above, on the efficiency of parsing.



$\}_1$	t: 45	b: 0	l: 100	r: 105		
	t: 45	b: 40	l: 0	r: 100		
$at\$3$	t: 40	b: 35	l: 20	r: 25		
$\Rightarrow$	t: 35	b: 30	l: 20	r: 23		
$\}_2$	t: 30	b: 0	l: 70	r: 75		
	t: 30	b: 25	l: 20	r: 30		
$at\$2$	t: 25	b: 20	l: 30	r: 35		
	t: 20	b: 15	l: 30	r: 55		
$arriveup$	t: 15	b: 10	l: 55	r: 70		
	t: 10	b: 5	l: 30	r: 55	p: 40	
$\neg at\$2$	t: 5	b: 0	l: 40	r: 47		
L_1	t: 45	b: 40	l: 0	r: 100	m: 20	
F_2	t: 40	b: 35	l: 20	r: 25		
Ai_1	t: 30	b: 25	l: 20	r: 30		
F_6	t: 25	b: 20	l: 30	r: 35		
Q_1	t: 30	b: 20	l: 20	r: 35	m: 30	
Av_1	t: 20	b: 15	l: 30	r: 55		
F_7	t: 15	b: 10	l: 55	r: 70		
Qv_1	t: 20	b: 10	l: 30	r: 70	m: 55	
Qq_1	t: 30	b: 10	l: 20	r: 70	lm: 30	rm: 55
L_2	t: 10	b: 5	l: 30	r: 55	m: 40	
F_4	t: 5	b: 0	l: 40	r: 47		
F_3	t: 30	b: 0	l: 20	r: 75		
F_1	t: 40	b: 0	l: 20	r: 75		
F	t: 45	b: 0	l: 0	r: 105		

Figure 12: Parse tree for ArriveDown