

Requirements Capture with RCAT

Margaret H. Smith
Klaus Havelund

*Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109, USA
margaret@jpl.nasa.gov
klaus.havelund@jpl.nasa.gov*

Abstract

NASA spends millions designing and building spacecraft for its missions. The dependence on software is growing as spacecraft become more complex. With the increasing dependence on software comes the risk that bugs can lead to the loss of a mission. At NASA's Jet Propulsion Laboratory new tools are being developed to address this problem. Logic model checking [9] and runtime verification [5] can increase the confidence in a design or an implementation. A barrier to the application of such property-based checks is the difficulty in mastering the requirements notations that are currently available. For these techniques to be easily usable, a simple but expressive requirement specification method is essential. This paper describes a requirements capture notation and supporting tool that graphically captures formal requirements and converts them into automata that can be used in model checking and for runtime verification.

1. Introduction

Defects can be introduced in all phases of software development, from requirements to software maintenance. It is desirable to intercept defects as early as possible in the development process, well before traditional software testing begins. The reliance on informal statements of requirements leaves the software development process vulnerable to a large class of potential defects caused by ambiguity and incompleteness in requirements. Consequently,

current methods of software testing provide relatively poor coverage of the original requirements and provide little control over what is tested.

In the current development process of robotic space missions at NASA's Jet Propulsion Laboratory (JPL), usually the set of requirements for a project are developed and maintained informally in the form of Word documents. The result of this lack of formality is a set of requirements that can have significant gaps and ambiguities. Furthermore, the relation between requirements and testing becomes informal. Informal text does not support test case generation, nor monitoring of requirements during system execution.

Part of the process for development of complex software at JPL is to create abstract *designs* in the modeling language Promela of the SPIN model checker [9], and check that these models satisfy various temporal requirements, usually referred to as *properties* [7]. Such properties of the Promela models are typically either stated in Linear Temporal Logic (LTL) [12], or as Büchi automata [3]. Büchi automata (never claims) form the foundation of property specification in SPIN in the sense that LTL is translated to these. However, neither LTL nor Büchi automata are specifically user-friendly notations for writing *complex* properties. To address this, JPL has developed a tool called the Requirements Capture and Analysis Tool (RCAT). The tool is mainly intended to support specification of behavioral requirements of Promela design models. An RCAT model can be automatically converted into a Büchi automaton for use by SPIN, and can hence be used to state a property about a Promela design model. Furthermore, monitors can be generated from RCAT that can be fed into the

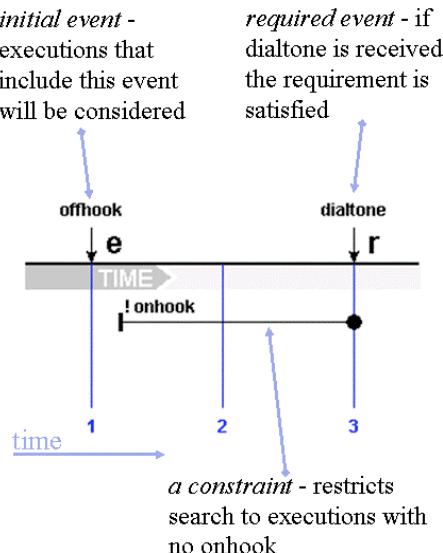
RMOR monitoring tool [6], which can perform monitoring of C programs against the properties during execution.

The RCAT graphical notation is based on state machines, with additional notation for expressing liveness properties (that some event must eventually happen), which are used frequently in model checking. An important objective for the RCAT tool is that it should require minimal training so that it will be easy for both systems engineers as well as software engineers to use it. The notation offers a total of only 7 graphical symbols. The tool interface closely resembles a Powerpoint chart but with a smaller palette of drawing features. RCAT utilizes state machines which form a common reference of understanding across educational boundaries. Note, however, that state machines here are used for expressing requirements and not designs. As will be outlined, the design of RCAT is responsive to experiences with and reflections on building or applying other requirements capture notations at JPL, such as the timeline editor [14], and the already mentioned LTL, and Büchi automata.

2. Notation Design Principles

Logic model checking tools commonly use a temporal logic to express requirements, which include concepts of temporal ordering, allowed, required and prohibited behaviors. The SPIN model checker, as already mentioned, uses LTL and Büchi automata.

A positive statement of the requirement that response to an offhook is dialtone



Such notations can be convenient in certain cases, but are generally hard to master by non-experts. Subtle errors in logic expressions may also be hard to identify and may lead to false positives and false negatives during verification and testing. This observation lead (prior to RCAT) to the design of the Timeedit tool [14], shown in Figure 1, where the fundamental concept is that of a time line upon which events are laid out. A timeline is a graphical depiction of a progression in time, stating behavior requirements that can be used to create formal properties for the SPIN model checker.

In the telecommunications domain for which the notation was designed, timelines are able to express many properties of interest. In this domain most properties include an event preamble (a sequence of events that must happen before a response is required), a trigger event, a response, and a set of constraints that if not met, will discharge the property. In addition to the benefits of the timeline notation for expressing properties in the telecommunications domain, it has these general advantages:

- *Simple* – the notation is easy to learn and use.
- *Intuitive semantics* – there is an obvious relationship between a timeline and its corresponding Büchi automaton.

While timelines have advantages, they are, however, not sufficient for the following reasons:

- *Expressiveness* – timelines cannot express iterative behaviors, and generally do not have the same expressive power as LTL, which in turn is less expressive than Büchi automata.
- *Cohesiveness* A single timeline can only express one requirement. It cannot express a collection of requirements. This is not a problem with expressiveness since several timelines can be used, but it causes a proliferation of timeline diagrams, resulting in very large specification documents.

The RCAT tool aims to combine the graphical intuition of an event timeline with additional expressiveness, to produce a tool that is still easy to use by non-experts in formal verification, yet can express a useful range of verifiable software design requirements.

Figure 1. Time line notation of the Timeedit

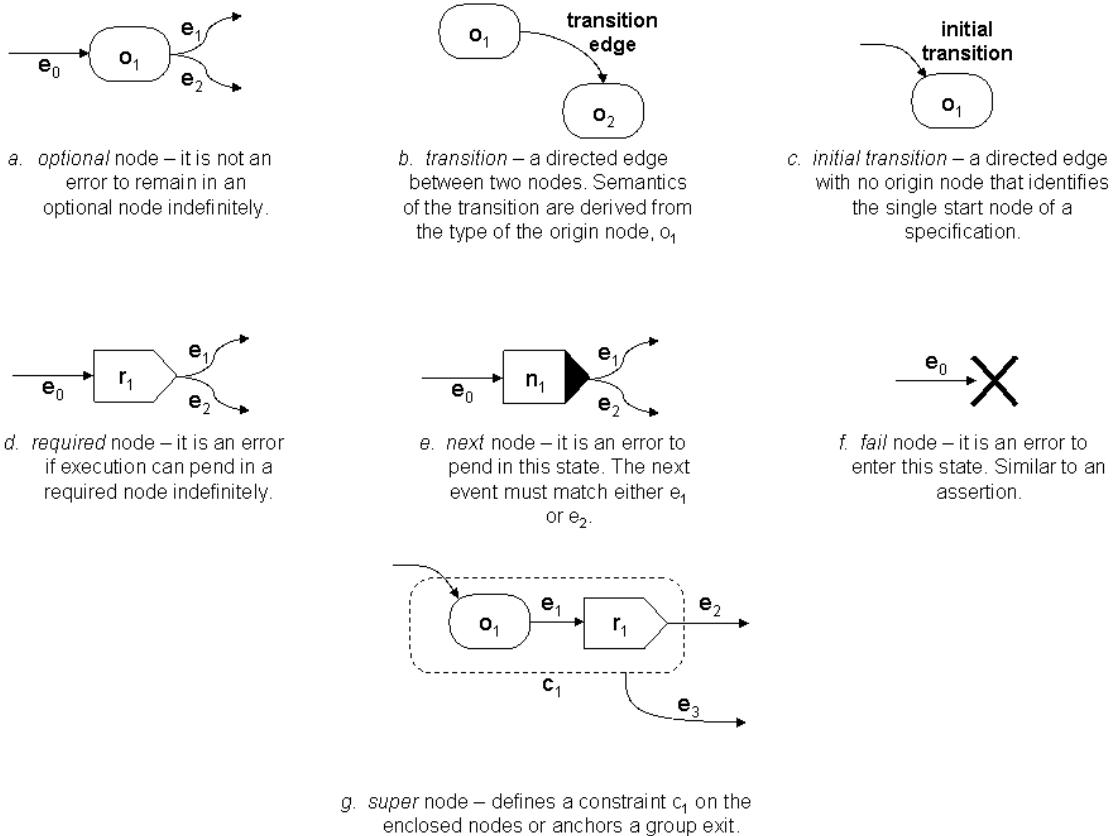


Figure 2. RCAT notation

RCAT supports a “two-dimensional” time line notation, allowing the time line to branch out in a tree-like format. Consider for example the property: “in case an event e_1 occurs, then either e_2 and e_3 must occur, in that order, or e_4 and e_5 must occur, in any order.” This type of property cannot be expressed in the timeline notation. To capture a sufficiently broad range of design requirements we must be able to express the following:

- the temporal ordering of events
- general Boolean constraints on executions, e.g. to limit the domain of interest to only specific types of executions that satisfy the constraints
- failure (error) events
- optional events that do not have to occur, but if they occur they may lead to other requirements that other events must occur
- required events that must eventually occur

- immediate events that must occur in the next execution step
- branching: choices between alternative event sequences

A solution would be to use Büchi automata directly to write requirements. However, Büchi automata are usually regarded as non-intuitive and complicated to write. Even experts can have problems writing such automata and getting them right. As it turns out, a slight modification of the Büchi automata notation yields a very practical yet simple and intuitive notation, which is the RCAT notation.

3. The RCAT Notation

RCAT is fundamentally a state machine notation with five different state symbols and two transition symbols, shown in Figure 2. The state symbols, which can be thought of as nodes in a (possibly cyclic) graph, are connected by directed edges: transitions. A transition may have a text label, defining an

observable *event in a system execution*. An *event* can be a system action or the truth of a Boolean expression on system state variables. If the event or satisfying system state occurs, the transition is enabled and control can pass to its destination node. Each RCAT specification has a single *initial transition*. The *initial transition* has no origin node. Its destination node is called the *initial node*. Any node type, except a *super node* (to be defined below) and the fail node, can be designated as the initial node of an RCAT specification. There are five types of nodes in RCAT (Figure 2): *optional*, *required*, *next*, *fail* and *super nodes*.

Optional node An *optional* node is drawn as a rounded box with a text label. Execution can wait in an optional node, awaiting one of the events on the outgoing transitions. It is not an error to wait in an optional node indefinitely. Text inside the box has no formal semantics: it is a comment only that can be used to increase the understandability of the diagram.

Required node A *required* node is drawn as a rectangle with a pointed side. Execution waits in a required node, awaiting one of the events on the outgoing transitions. It is an error if one of these events does not occur eventually.

Next node A *next* node is drawn as a rectangle with a black triangle on one side. Execution cannot wait in a next node. If the immediately next event within the scope of the RCAT specification matches one of the outgoing transitions, execution moves to the matching destination node. It is an error if the next event does not match one of the outgoing transitions.

Fail node A *fail* (or *error*) node is represented by an ‘X’. It is always an error to enter a fail node. It is not possible to leave a fail node once it is entered.

Super node A *super node*, or *group node*, is drawn as a rounded box with a dashed line. The main purpose for the super node is to define a constraint on the behavior that is captured by the nodes that are enclosed. The constraint, designated in Figure 2.g. as c_1 , indicates that the events and states in the super node are only relevant while constraint c_1 evaluates to true. The constraint applies to all transitions that have their source node inside the super node. In the example in Figure 2.g., e_1 and e_2 are constrained by c_1 . If for an execution under consideration the constraint ceases to be true, it is not an error. On the contrary, the execution is no longer of interest because

failure of the constraint to hold means that the execution cannot contain an error. A *super node* can also be used to anchor a group exit, such as transition e_3 on Figure 2.g. A group exit is used to indicate that while execution is pending in any of the nodes within the super node, if the matching event e_3 occurs, the e_3 transition will be taken. A constraint, if present, will also constrain any group exit transitions.

Semantics of initial node The final element of the RCAT notation determines the semantics of the initial node. The user can select *once* or *everytime* semantics (in the graphical editor tool menu). A *Once* semantics defines a single check for the *first* occurrence of any one of the events on transitions that leave the initial node. An *Everytime* semantics defines a check for *every* occurrence of any one of the events on the transitions that leave the initial node, defining a stronger check. An example of a requirement where *everytime* semantics would be appropriate is

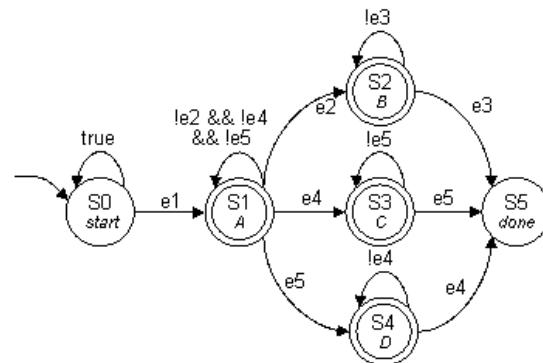
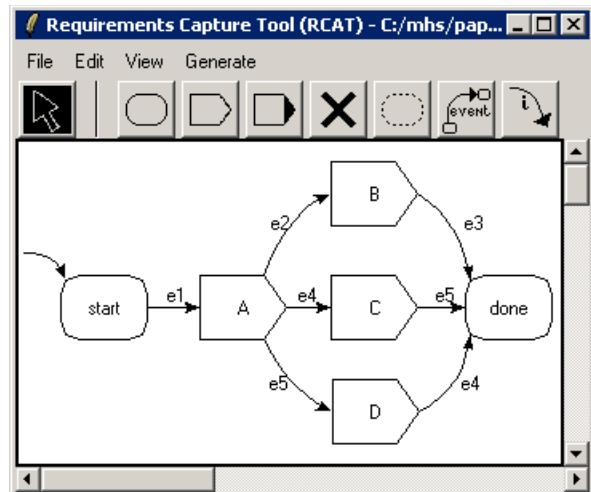


Figure 3. A property in the RCAT notation and the correponding Büchi automaton

“whenever communication with ground is requested the rover should eventually stop”. An example of a requirement where *once* semantics could be used is: “once the spacecraft has entered the atmosphere, the main parachute is deployed”. The latter is only expected to occur once, while communication with ground is a repeated activity, and each time the rover should stop (yielding power resources to communication).

A simple RCAT property is depicted in the upper half of Figure 3. This RCAT model captures the property we considered earlier: “in case an event e_1 occurs, then either e_2 and e_3 must occur, in that order, or e_4 and e_5 must occur, in any order”. Following the initial transition the initial node, informally labeled “start,” is entered. Since this node is an *optional* node, we can stay in this node indefinitely without it being considered an error. However, once event e_1 is detected, control may be passed to *required* node A. It is an error to remain in node A indefinitely, meaning that it is an error if one of e_2 , e_4 or e_5 is not eventually received. Similarly, since node B is a *required* node it is an error if e_3 does not eventually occur. Similar logic applies to *required* nodes C and D. Once control is passed to *optional* node “done” no further errors can potentially be detected and the verifier will stop checking the execution.

RCAT is meant to be a specification language for the SPIN model checker, in which properties about Promela models can be stated. In order for SPIN to verify that a Promela model satisfies a temporal property, the property must be represented as a so-called never claim, essentially a Büchi automaton. The RCAT tool offers a translator from the RCAT notation to Büchi automata. The lower half of Figure 3 shows the Büchi automaton generated from the just described RCAT automaton. For the reader not familiar with Büchi automata, the following brief explanation should suffice. The language of a Büchi automaton is

node type	optional	required	next
RCAT notation			
automaton			

Figure 4. Conversion of RCAT specifications to Büchi automata

a set of infinite traces, each trace corresponding to a path through the automaton making transition conditions over events evaluate to true. A Büchi automaton has two kinds of states: normal states (drawn as a single circle) and acceptance states (a double circle). An infinite trace is in the language of a Büchi automaton (is accepted by the automaton) if it visits an acceptance state infinitely often. It is implied that precisely one transition must be taken within the automaton at *each* model execution step. At a given automaton state, if there is no matching transition that can be taken, then the current execution is not matched by the automaton and the remainder of the execution is considered to fall outside the scope of the automaton (is not accepted by the automaton). Checking a property of a Promela model in SPIN conceptually corresponds to checking that the language denoted by the model forms a subset of the language denoted by the property. This is, however, in practice done by checking that the intersection of the language denoted by the model with the complement of the language denoted by the property, namely the language of bad traces, is empty. Hence, in SPIN, a Büchi automaton used for verification, also referred to as a *never claim*, must accept all bad traces.

4. Conversion to Büchi Automata

The first step in generation of a Büchi automaton from an RCAT automaton is to convert every transition leaving a *super* node, to the individual transitions from sub-nodes that the super node transition represents. Each node in the RCAT specification is subsequently converted into a Büchi automaton state, with the exception of the *super* nodes. The automaton states generated for *optional* and *required* nodes are respectively normal states and acceptance states. The following description will focus on the optional node o_1 in Figure 4. The self-loop on the corresponding automaton state s_1 is labeled with the conjunction of the negation of all transition labels sourcing node o_1 and all constraints on super nodes that overlap any transition sourcing o_1 . A transition that sources the automaton state s_1 is created for each transition sourcing o_1 . A label for each transition sourcing the automaton state is derived from the corresponding RCAT model transition label and the conjunction of the super node constraints that overlap the source of the RCAT model transition.

For the optional node, as long as we do not receive one of the events that originates at the node and all constraints hold true, we can follow the self-loop transition on the state s_1 . s_1 is a non-accepting state so it is not an error to take the self-loop infinitely often. Nor is it an error for a constraint to become false. In this case the execution is discarded because it is no longer of interest (cannot potentially contain errors).

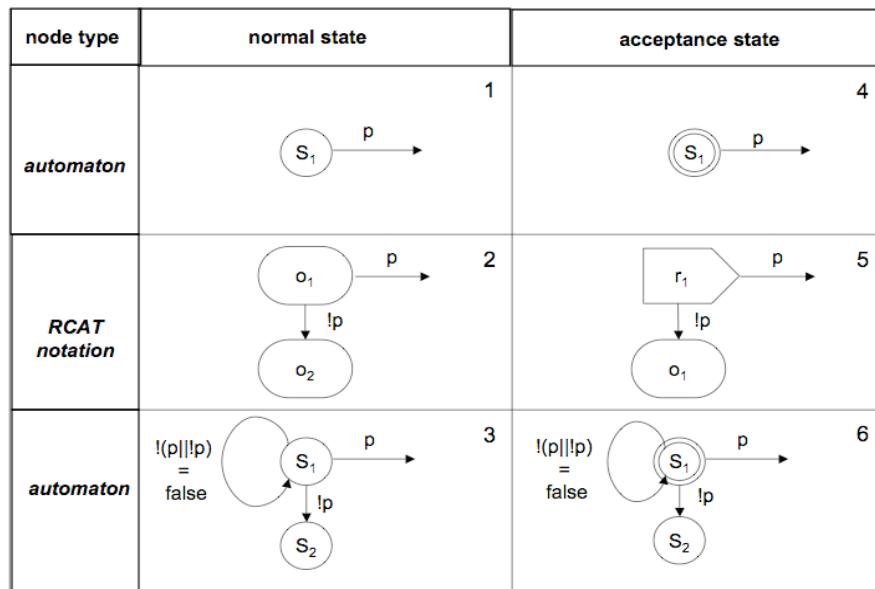


Figure 5. Mapping Büchi automata to RCAT and back

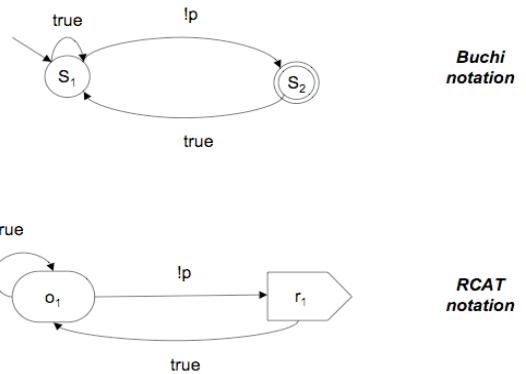


Figure 6. Negation of $\lhd \lceil p$ as Büchi automaton and RCAT automaton

For the required node, execution also remains in the state s_1 until one of the events that originates at the node occurs. However, since s_1 is now accepting, it is an error if we take the self-loop on s_1 infinitely often.

The automaton state corresponding to a next node has no self-loop, meaning that it is not possible to stay in this state for more than one step. For each transition sourcing next node n_1 , a transition is created sourcing the corresponding automaton state s_1 and labeled with the transition's event label and the conjunction of any overlapping constraints from super nodes. These transitions represent the desired next events and are the only means of escape from the state. A single additional transition sources the node s_1 to reach a

state s_2 in the case where none of the required events occur in the next execution step. This transition is labeled with the conjunction of the negated labels of each transition sourcing the next node n_1 and the conjunction of any overlapping super node constraints.

For each fail node in the RCAT specification, an accepting state with a self-loop labeled *true* is created in the automaton. For each incoming transition to the RCAT fail node, an incoming transition with the same label is created to this corresponding accepting state.

The labels that appear in the RCAT specification can

be linked to Promela Boolean expressions in the RCAT *dictionary*. That is, in the automaton that is generated by RCAT, a symbol is generated for each such label, and the connection between the symbol and a Boolean expression is established by a SPIN macro definition.

5. Expressiveness

The timeline notation is strictly less expressive than LTL, which again is strictly less expressive than Büchi automata. RCAT, however, has the same expressive power as Büchi automata for infinite traces, and hence consequently allows to state any property that can be stated in the timeline notation or in LTL, as well as in Büchi automata. Convenience is another matter, which we shall return to.

To show that RCAT has the same expressive power as Büchi automata, we have to show that (i) for every RCAT automaton there exists a Büchi automaton that accepts the same language, and (ii) for every Büchi automaton there exists an RCAT automaton that accepts the same language. Direction (i) is obvious since RCAT (in this paper) is given semantics by translation into Büchi automata. Direction (ii) is somewhat obvious, but requires a little explanation. What needs to be proved is that for every Büchi automaton B , there exists an RCAT automaton R that denotes (translates into) a Büchi automaton B' that is equivalent to the original Büchi automaton B (accepts the same language). The argument for (ii) is outlined informally by showing that for each single arbitrary Büchi automaton state (normal or acceptance) with an exiting transition, there is an RCAT state with “equivalent semantics”.

A normal Büchi state and an acceptance Büchi state are illustrated by Figure 5, fields 1 and 4. Consider the normal state in field 1. It illustrates a state in which the property p has to be true in the next step, otherwise the automaton that contains this state will block (unless there are other transitions enabled). A corresponding RCAT automaton is shown in field 2, which in turn is translated into the Büchi automaton in

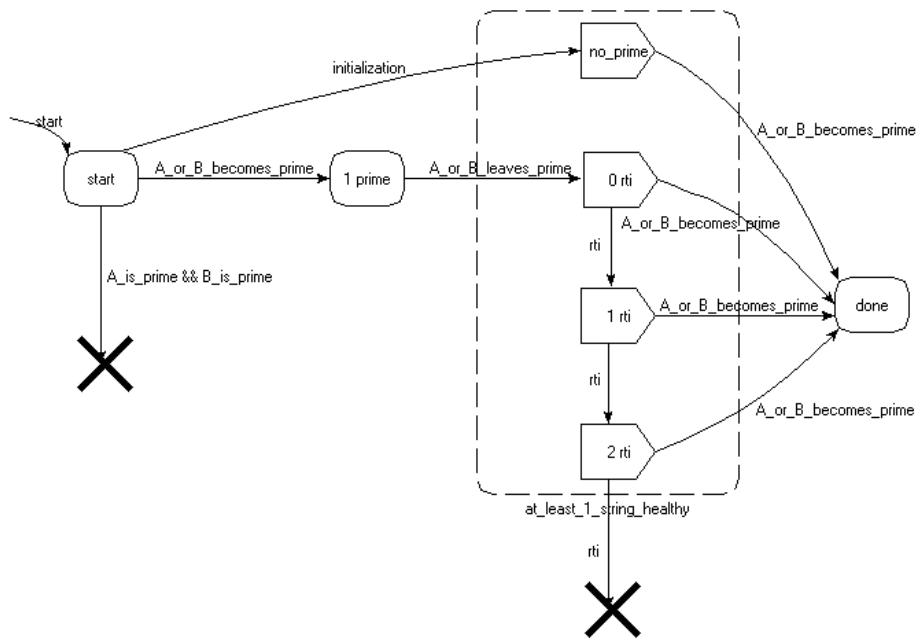


Figure 7. Dual string arbitration requirements specified in RCAT

field 3. The latter is equivalent to the automaton in field 1. This is because transitions that are false or which end in states with no exit transitions can be eliminated in a Büchi automaton, while preserving the semantics. The same reasoning holds for acceptance states, see fields 4, 5 and 6. Note that this theoretical argument may give the impression that RCAT is more verbose than Büchi automata (requiring more symbols). However, it is generally the other way around for practical purposes.

The equivalence of Büchi automata and RCAT can be illustrated by the LTL property “ $\langle\!\rangle[]p$ ”, which states that eventually ($\langle\!\rangle$) a state must be reached, where the property p becomes stable always ([] true).

To check this property against a Promela model in SPIN we have to convert its negation “ $!\langle\!\rangle[]p$ ” into a Büchi automaton, and analyze the product of this negation with the Promela model. The negated automaton will then “accept” any infinite trace that violates the original property “ $\langle\!\rangle[]p$ ” by visiting an acceptance state infinitely often. The negated formula “ $!\langle\!\rangle[]p$ ” translates into the Büchi automaton shown at the top of Figure 6. This automaton accepts any word where $!p$ is true infinitely often, which essentially means that p never becomes stable (always true). The equivalent RCAT automaton is also shown at the bottom of Figure 6, generated using the principles outlined in Figure 5. The idea is that in case of an infinite trace that violates “ $\langle\!\rangle[]p$ ”, it will hold that $!p$

is true infinitely often, which again means that state r_1 in the RCAT automaton is visited infinitely often, which signals an error since it is mapped to an acceptance state. In this case the RCAT automaton becomes almost identical to the Büchi automaton.

The example illustrates how loops involving *required* states in RCAT can be used to express certain properties. Note, that this effect of loops involving *required* states may come as a surprise to users familiar with normal state machines as found for example in UML, where looping is the normal way of modeling iteration. The normal use of RCAT should be to create non-looping state machines.

Another example of a class of properties that become difficult to express in RCAT (although possible) are fairness properties, such as for example the following LTL property:

$$(\Box \Diamond p) \rightarrow \Box(q \rightarrow \Diamond r)$$

It states that if p is true infinitely often, then it holds that any occurrence of q eventually results in an occurrence of r . Each of the two component properties of the outer implication operator, hence " $\Box \Diamond p$ " and " $\Box(q \rightarrow \Diamond r)$ ", can easily be stated as RCAT automata individually, but the combination cannot be easily stated, although it can be stated since RCAT is as expressive as Büchi automata. Unfortunately, the generation of Büchi automata from LTL is not compositional and therefore the automaton for the formula above cannot be generated from the automata from the components.

Note that RCAT's *required* node corresponds to LTL's until operator, and RCAT's *next* node corresponds to LTL's next operator. LTL's next operator is normally regarded as dangerous and it is usually recommended to avoid the use of this operator. The next node can, however, be considered useful when there is a need for specifying what

should happen for each event in a finite set of events, where it is part of the requirement that no other event can happen.

6. Application of RCAT

A prototype version of the RCAT tool was implemented in approximately 6,000 lines of Tcl/Tk [11]. The following example illustrates the application of RCAT to a flight hardware redundancy algorithm, referred to as the *dual-string* algorithm, developed at JPL in a recent spacecraft development project. The dual string arbitration algorithm is heritage from the Cassini spacecraft [13], but has been re-engineered to utilize the current generation avionics platform. Because of the large number of signal and state combinations to be tested in this system, model checking was selected to perform an exhaustive coverage of the design space. The algorithm was modeled in Promela, and the properties were verified in SPIN.

Hardware redundancy is a strategy used to ensure

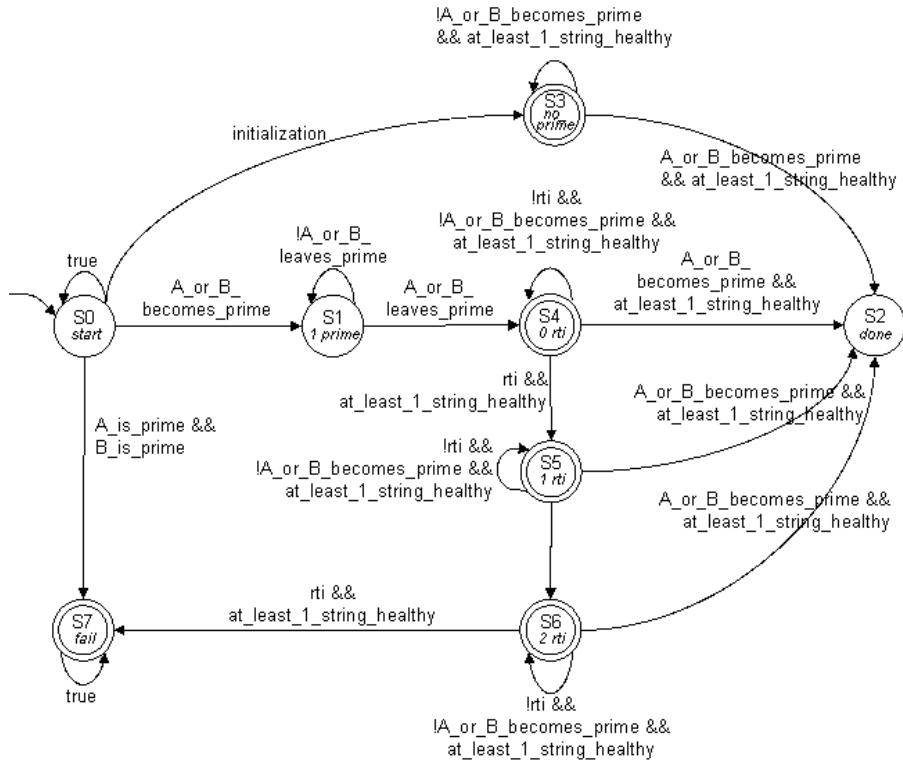


Figure 8. Büchi automaton for the dual string arbitration requirements

```

[](init -> <>(A_or_B_becomes_prime | !at_least_1_string_healthy))
&
[]!(A_is_prime & B_is_prime)
&
[](A_or_B_becomes_prime & <>A_or_B_leaves_prime) ->
(!A_or_B_leaves_prime U X (A_or_B_leaves_prime &
(!rti U (A_or_B_becomes_prime | !at_least_1_string_healthy)) |
(!A_or_B_becomes_prime & at_least_1_string_healthy) U
( rti & X (!rti U (A_or_B_becomes_prime | !at_least_1_string_healthy)) |
(!A_or_B_becomes_prime & at_least_1_string_healthy) U
( rti & X (!rti U (A_or_B_becomes_prime | !at_least_1_string_healthy))))))))

```

Figure 9. LTL formula for dual-string arbitration requirements

uninterrupted operation of a spacecraft in the face of unrecoverable hardware faults. A compute unit (referred to as a *string*) and its associated boards is “cross-strapped” to an identical copy across a dual-redundant spacecraft bus. In operating mode, one compute unit is in control, or is *prime* while the other waits in a quiescent state known as *backup*. State information passes from the *prime* to the *backup* at each real time interval (RTI) to update the state of the *backup*.

When the spacecraft is initially powered on, an arbitration algorithm executing on both compute units is used to negotiate which compute unit takes the *prime* role and which becomes the *backup*. The compute unit playing the *prime* role controls the spacecraft radio and has access to the *prime* spacecraft bus for issuing commands to non-redundant components such as instruments and the power system. Cross-string signal connections exist between the two compute units to permit direct monitoring by one compute unit of the *prime* or *backup* state that is being asserted by the other unit. If a fatal fault occurs on the compute unit that is in the *prime* role, it de-asserts primeness on its outgoing cross-string signals, and the redundant compute unit then has an opportunity to advance to the *prime* role.

This brief introduction to the dual-string arbitration algorithm omits many essential details that add complexity and make verification non-trivial. For instance, extra states in addition to *prime* and *backup* are supported as well as redundancy in individual cross-string signals. Also, there are command-controlled switches that allow the ground operations team to intervene by enabling and disabling specific arbitration behaviors. A test harness encompassing all combinations of compute unit states, cross-string signal states, and command control states would consist of millions of test cases and would not be feasible to conduct on flight hardware. The Promela model attempts to explore the state space of an abstraction of this system. To perform the verification,

the following key requirements were formalized, originally using LTL and Promela inline assertions:

“When the spacecraft is initially powered on, one compute unit must become prime.”

“Under the assumption that the spacecraft is in a healthy state, no more than one compute unit should ever control the spacecraft, hence be in the prime state.”

“In the case of a single fault in a compute unit, the other compute unit should enter prime state in less than 3 RTIs.”

Assuming that the compute units are identified in hardware as “A” and “B,” an RCAT model corresponding to these requirements is shown in Figure 7. An *everytime*-semantics is selected for this chart, meaning that the properties specified in this chart must be satisfied *whenever* the exit conditions of the start node are matched. A graphical depiction of the generated Büchi automaton is shown in Figure 8. The formulation of this property in LTL is shown in Figure 9. In the LTL version, the three requirements are composed by conjunction. The first two requirements have a fairly straightforward representations in LTL. The third requirement demonstrates that LTL does not provide a simple form for capturing properties containing sequences of events.

Using the SPIN model checker, the formalized RCAT requirement can be verified against the Promela model. Several important close calls and a single design flaw were identified by SPIN during the initial verification effort. One of these design flaws was a confirmation of a hardware design flaw that resulted in a hardware re-design. Details of this work are presented in [7].

7. Related Work

RCAT is an improvement of the Timeedit tool, in the sense that it adds conditional branching, cycles and a next node. RCAT has the same expressive power as Büchi automata, and can alternatively be seen as a user-friendly version of Büchi automata, focusing on a particular set of properties which become natural to express in this notation. The main difference from Büchi automata is the lack of need for defining self-

loops in many practical cases. RCAT is essentially state machines, with the notion of “liveness” states, which have to be left once entered. One can imagine an extension of UML statecharts with this notion of liveness, and in fact we are considering defining RCAT as a UML profile. This would have the advantage that all of UML’s infrastructure would become available. Visual Timed Event Scenarios (VTS) [2], which also builds on Timeedit, permits conditional branching, cycles and annotation of diagrams with timing constraints for real-time model checking, but omits constructs for expressing next and fail events. RCAT is suited for specifying many of the specification patterns defined in [1]. These patterns express properties that must hold in scopes, such as “*the property P should become true between Q and R*”. The property of the dual string algorithm is an example of such a property, the LTL equivalent of which is rather complicated.

8. Conclusions

The RCAT tool was developed to fill a gap in the area of formalizing behavioral design requirements for mission critical software applications. The goal of this tool is to preserve the simplicity of notation and ease of use of the predecessor tool, Timeedit, while providing some essential improvements in expressiveness and convenience. The RCAT tool is currently being studied at JPL for infusion into mainstream use. RCAT appears sufficient for expressing properties of Promela models. For runtime verification of executing code the notation needs to be further augmented with data values to be fully effective. Further work includes defining the RCAT notation as a Unified Modeling Language profile.

Acknowledgements

RCAT was developed at JPL, California Institute of Technology, under the Reliable Software Systems Development project, using National Aeronautics and Space Administration (NASA) funds administered through the NASA Exploration Systems Missions Directorate. JPL employees Martin Feather, Rajeev Joshi, Alex Groce, Gerard Holzmann, and Nicolas Rouquette provided valuable input.

References

- [1] H. Alavi, G. Avrunin, J. Corbett, L. Dillon, M. Dwyer, and C. Pasareanu, Specification Patterns. SAnToS

Laboratory, Kansas State University,
<http://patterns.projects.cis.ksu.edu>.

- [2] A. Alfonso, V. Braberman, and N. Kicillof, Visual Timed Event Scenarios, Proc. 26th Int'l Conf. on Software Engineering (ICSE'04), May 23-28, 2004, Edinburgh, Scotland, pp. 168-177.
- [3] J. R. Büchi, On a Decision Method in Restricted Second Order Arithmetic. Proceedings of the International Congress on Logic, Methodology and Philosophy of Science, Stanford, pages 1-11, Stanford Univ. Press, 1960.
- [4] D. Drusinsky, Modeling and Verification using UML Statecharts. Elsevier, 400 pages, ISBN-13: 978-0-7506-7949-7, 2006.
- [5] K. Havelund and G. Rosu, Synthesizing Monitors for Safety Properties, Tools and Algorithms for Construction and Analysis of Systems, TACAS 2002, April 6-14, 2002, Grenoble, France, Vol. 2280, pp. 342-356.
- [6] K. Havelund, Runtime Verification of C Programs. In Proc. of the 20th IFIP Conf. on Testing of Software and Communicating Systems (TESTCOM / FATES'08), LNCS 5047, pp. 7-23, Springer. Tokyo, Japan, June 2008.
- [7] K. Havelund, G. Holzmann, and M. Smith. Verification of a String Arbitration Algorithm. Jet Propulsion Laboratory, internal report. October 2007.
- [8] K. Havelund, M. Lowry, and J. Penix, Formal Analysis of a Space Craft Controller using SPIN, IEEE Trans. on Software Engineering, Vol. 27, No. 8, August, 2001.
- [9] G. Holzmann, The Model Checker Spin, IEEE Trans. on Software Engineering, Vol. 23, No. 5, May 1997, pp. 279-297.
- [10] G. Holzmann, The SPIN Model Checker: Primer and Reference Manual, Addison-Wesley Professional, September, 2003. ISBN 0321228626.
- [11] J. K. Ousterhout, Tcl and the Tk Toolkit, Addison-Wesley Professional; 1st edition, March 31, 1994. ISBN 020163337X.
- [12] A. Pnueli, The Temporal Logic of Programs. Proceedings of the 18th IEEE Symposium on Foundations of Computer Science, pages 46-77, 1977.
- [13] F. Schneider, S. M. Easterbrook, J. R. Callahan, and G. J. Holzmann, Validating Requirements for Fault Tolerant Systems using Modeling Checking, Proc. int. Conf. on Requirements Engineering (ICRE), pp. 4-14, IEEE, Colorado Springs, CO. USA, April 1998.
- [14] M. Smith, G. Holzmann, and K. Ettessami, Events and Constraints: a Graphical Editor for Capturing Logic Properties of Programs, 5th Int'l Sym. on Requirements Engineering, pp 14-22, Toronto, Canada. August 2001.