Two Decades of Industrializing Formal Verification: The Reactis Story*

Rance Cleaveland[1][0000−0002−4952−5380], David Hansel[2], Steve Sims[2], and Scott A. Smolka[3][0000−0002−7348−630X]

[1] Department of Computer Science, Univ. Maryland, College Park MD 20742, USA
`wcleavel@umd.edu`

[2] Reactive Systems Inc., 341 Kilmayne Dr. Suite 101, Cary NC 27511, USA
`{hansel,sims}@reactive-systems.com`, `www.reactive-systems.com`

[3] Department of Computer Science, Stony Brook Univ., Stony Brook NY 11794, USA
`sas@cs.stonybrook.edu`

Abstract. Reactis® is a suite of tools produced by Reactive Systems, Inc. (RSI), for automated test generation from, and verification of, systems given in either the modeling languages MATLAB® / Simulink® / Stateflow® of The MathWorks, Inc., or ANSI C. RSI was founded by three of the authors of this paper in 1999, with the first release of Reactis coming in 2002; the tools are used in the testing and validation of embedded control systems in a variety of industries, including automotive and aerospace / defense. This paper traces the development of the Reactis tool suite from earlier research on model-checking tools undertaken by the authors and others, highlighting the importance of both the foundational basis of Reactis and the essential adaptations and extensions needed for a commercially successful product.

Keywords: Reactis · Model-based testing · MATLAB / Simulink / Stateflow · Guided simulation · Instrumentation-based specification and verification

1 Introduction

Reactis®[4] [43] is a suite of tools for the validation of embedded control systems. The tool suite provides two key functionalities:

1. automated test generation from The Mathworks Inc.'s MATLAB® / Simulink® / Stateflow® (hereafter referred to collectively as "Simulink") modeling environment[5] models and ANSI C code; and
2. verification of user-specified requirements against models / code.

* Supported by Small Business Innovation Research (SBIR) funding from the US National Science Foundation and Office of Naval Research.

[4] Reactis® is a registered trademark of Reactive Systems, Inc.

[5] MATLAB ®, Simulink® and Stateflow®are registered trademarks of The MathWorks, Inc.

Reactis is developed and sold by Reactive Systems., Inc., a company founded by three of the authors of this paper (Cleaveland, Smolka and Sims) in 1999; the fourth author (Hansel) was the first non-founder technical employee of the company. The tools' primary user base consists of engineers in the automotive and aerospace / defense industries, where the use of Simulink and C is ubiquitous in the design and development of embedded control systems. Customers include companies in over 20 countries world-wide.

While targeting testing-based-validation use cases, Reactis has its intellectual foundations squarely in the formal-verification research community, and more specifically, in model checking [2, 11, 27]. In particular, the techniques Reactis uses to generate thorough suites of test cases from models / code are heavily influenced by the state-space search methodologies used in *on-the-fly* model checking [7, 12, 26, 48] (which is also sometimes referred to as local or tableau-based model checking). They also employ constraint-solving procedures from Satisfaction Modulo Theories (SMT) [3, 10, 36] tools. This foundational basis is fundamental to the quality of Reactis' performance, although other aspects of the tools' functionality have played essential roles in its success in the marketplace.

This paper traces the development of Reactis, from its formal-verification roots, to its initial market-driven design, to its subsequent and ongoing development. In particular, we highlight how the tools' grounding in formal methods has been a key factor in their eventual uptake, but this basis would not have been sufficient on its own; other concerns have also been critically important. The remainder of this paper is structured as follows. The next section describes the origins of the Reactis tool, and in particular the research efforts of the RSI founders before the company's launch that informed the design and development of Reactis. Section 3 reviews the launch of RSI, the initial vision of the founders for the company as a developer and purveyor of model-checking tools, and a crucial strategic pivot made early in the life of the company to focus on Simulink. The section following discusses the initial design decisions of Reactis and their relation to the founders' choice to focus on Simulink. Section 5 focuses on the commercial considerations inherent in obtaining customers for Reactis, and how these considerations influenced subsequent additions and modifications to the Reactis tool suite. Section 6 gives a snapshot of the current Reactis tools, while the final section offers our concluding remarks and perspectives on the future.

2 Origins of Reactis: Formal Methods / Model Checking

This section describes the intellectual foundations of Reactis, with a focus on research by the company founders (Cleaveland, Sims and Smolka) that laid the groundwork for the launch of Reactive Systems Inc. and the design and development of Reactive Systems' tools.

Cleaveland and Smolka have been active researchers in the formal methods community since the 1980s, both in collaboration with one another and with others. Smolka was an early pioneer in the development of verification algorithms based on bisimulation equivalence for finite-state systems [29, 30]. This style of verification has its roots in the process-algebra [5] community; it features the use of a (higher-level) system models as specifications for (lower-level) system implementations, with semantic relations such as bisimulation equivalence used to check conformance between specification and implementation. Cleaveland and others showed how bisimulation-equivalence and simulation-preorder [25] algorithms could be adapted to compute a variety of other semantic-refinement relations in process algebra [15]. Both Cleaveland and Smolka have also worked on model-checking algorithms for finite-state systems and temporal logics, with special attention to the *modal mu-calculus* [31], an expressive logic based on fixpoints that can encode all known temporal logics. In model-checking-based verification, specifications are given as formulas in a temporal logic, such as the modal mu-calculus; a system is deemed to satisfy the formula according to the logic's semantics. In the case of finite-state systems and the modal mu-calculus (as well as other temporal logics), this check can be decided algorithmically. Cleaveland and Smolka have each contributed efficient algorithms for various fragments of the modal mu-calculus [21, 33, 47] and have also designed strategies for parallelizing model checking [49] and achieving improved practical performance via translations of model checking into logic programming [41].

Of special note for the purposes of this paper is Cleaveland's and Smolka's work on so-called *on-the-fly* [26] methods (these methods have also been called *local* [48] and *tableau-based* [12]) for model-checking the mu-calculus [8, 12, 22, 32]. Traditional model-checkers compute the entire state space of a system, then check whether the states in the state space satisfy the temporal formula in question. In contrast, on-the-fly checkers generate states in a demand-driven fashion, based on the formula being checked, leading to more efficiency when not every state requires checking in order to determine the verification result. This approach also permits approximations to model checking to be performed by allowing some successor states to be omitted.

All four of the authors of this paper have developed research tools that implemented verification algorithms and other analyses for finite-state systems. Smolka and colleagues developed the Winston tool [34], which supported the graphical creation of hierarchical networks of processes and included capabilities for verification based on bisimulation and observational equivalence [35]. Cleaveland and colleagues created the Concurrency Workbench (CWB) [16], a tool for verifying systems described in Milner's Calculus of Communicating systems (CCS) [35]. A noteworthy feature of that tool was the compositional design of its analysis routines; a variety of semantic equivalences and preorders were implemented based on the combination of core equivalence routines combined with semantic transformations of the systems being analyzed. The tool also included a mu-calculus model checker. Cleaveland and Sims, whose PhD work was supervised by Cleaveland, redeveloped the Concurrency Workbench, yielding the Concurrency Workbench of North Carolina (CWB-NC) [18]. That tool refactored the design of the original CWB by decoupling the verification routines from the syntax and semantics of CCS. This enabled the development of a tool, the Process Algebra Compiler [13], that, given syntactic and semantic specifications of a modeling language, generates the necessary routines for building a CWB-NC that analyzes system descriptions given in that language. Sims and colleagues also worked on checking invariants of so-called Software Cost Reduction (SCR) tabular specifications of systems [24] (these specifications are sometimes called *Parnas* tables after their inventor [40]). The result of this work was the Salsa tool for checking invariants of SCR models [6]. Hansel's master's thesis was supervised by Cleaveland and Smolka, and focused on the generation of prototype implementations of distributed systems from validated formal specifications; this work is described in [23].

3 The Launch of Reactive Systems, and a Strategic Pivot

This section discusses the activities leading to the founding of Reactive Systems, and also describes a key turning-point in the company's strategy that led to the development of Reactis.

3.1 Lead-up to the Founding of Reactive Systems

As the previous section notes, the co-founders of Reactive Systems had extensive experience in the development of model-checking-based verifiers of finite-state systems, including both theory and tool development as well as distribution of these tools, primarily to other research groups. In the late 1990s, Cleaveland and Smolka began discussing the possibility of commercializing verification tools based on their research. The .com boom was in full swing, and both thought that, given the successful uptake of formal verification in the so-called Electronic Design Automation (EDA) marketplace for hardware design, and the dearth of verification tools targeting software design despite the evident need for improving software quality, there were market opportunities that could be pursued. With other colleagues they developed a hierarchical modeling notation and analysis tool, the Concurrency Factory [14], heavily influenced by the Winston tool [34], for the compositional modeling and verification of networks of finite-state machines. Their intention was to commercialize the Concurrency Factory; consequently, they applied for, and subsequently obtained, a patent for the concept [19]. They then began looking for investors for the following idea: a modeling and verification environment for concurrent / distributed software, with an initial focus on telecommunications companies. Also during this time, co-author Hansel arrived at Stony Brook University to work on his master's thesis under the supervision of Cleaveland and Smolka.

Meanwhile, after finishing his PhD studies at North Carolina State University Sims had taken a job as a research scientist at the US Naval Research Laboratory and was working on the Salsa tool [6] described above. Cleaveland and Smolka approached him to see if he might be interested in joining their commercialization effort; he agreed, and in 1999 the three incorporated Reactive Systems, Inc., still with the idea of developing and selling verification tools for concurrent and distributed systems based on the Concurrency Factory's bespoke modeling framework.

3.2 Early Reactive Systems: The Concurrency Factory

Once the company was launched, the founders needed capital to fund the development of the envisioned Concurrency Factory tool. In late 1999, they applied for a grant from the National Science Foundation (NSF) through the agency's Small-Business Innovation Research program, which provides research awards to small companies to help them commercialize basic research. They were successful in this endeavour; in June 2000, NSF awarded Reactis Systems enough funds beginning June 2000 for six months of exploratory research for the Concurrency Factory. The NSF grant also included a possibility for two years of additional follow-on funding if the results of the first six-month phase were promising.

An important meeting. The company also needed a business plan, which required market research, so the founders had begun attending industry-leaning conferences to gain insights into the prevailing commercial practices in software quality. At the 19th Digital Avionics Systems Conference (DASC) in October 2000, Sims met a key contact, an engineer from an automotive OEM[6]. The OEM engineer and Sims talked about the plans for Reactive Systems and the software-verification techniques of the Concurrency Factory, as well as about the software-validation techniques being used within the OEM. The engineer admitted that a tool like the Concurrency Factory would be a hard sell within his company, but that automated support for testing software against Simulink models would be of serious interest. The OEM at that time was using Simulink to design automotive subsystems, and there was an internal effort underway for using these models to guide the development of the embedded software to be run in these subsystems. Sims was intrigued, and the OEM engineer offered to send Reactive Systems a couple of their Simulink models to see what our analyses might infer about them.

This interaction between Sims and the OEM engineer, and the subsequent collaborative pilot study that ensued, had a couple of profound long-term impacts on the technical and business strategy of Reactive Systems.

First, the collaboration gave us our first experience working with automotive engineers who were developing embedded software applications. In the early 2000s, the use of software in vehicles was just beginning its rapid growth towards its current state, in which each new vehicle model contain dozens of microprocessors and millions of lines of code. To this day, automotive customers make up the largest component by far of Reactive Systems' customer base.

[6] In the automotive, and other, industries, companies are referred to as *Original Equipment Manufacturers*, or OEMs, if they produce end-of-the-line products for customers outside the industry. Car companies such as BMW and Toyota are OEMs, for example. There is terminology for suppliers in these industries as well. Tier-1 suppliers sell their products to OEMs, while Tier-2 suppliers sell to Tier-1 suppliers, etc. As an example, Robert Bosch is a Tier-1 automotive supplier that sells components such as anti-lock braking subsystems to automotive OEMs. Companies selling e.g. electronic components to Bosch would then be considered Tier-2 suppliers.

Second, the work with the automotive pilot study introduced us to Simulink as a modeling notation for designing and implementing automotive applications. As a startup with limited resources, the decision of which notation(s) our analysis tools would support was one of the most important choices we had to make. We also were in the midst of our initial NSF funding, and we had to decide whether to devote these resources to continuing to develop an initial prototype of the Concurrency Factory, which was based on our home-grown modeling notation, or to work with our automotive OEM collaborator on the Simulink-model analysis study. In the end we gambled that developing a relationship with a potential reference customer was a better way forward than purely focusing on developing our own modeling technology. Consequently, even though our familiarity with Simulink was non-existent (we did know about MATLAB, however), we elected to devote the remaining time of our initial NSF funding to pursuing the Simulink pilot study with our OEM partner.

Simulink pilot study. After Sims' return from DASC he contacted the OEM engineer, and they defined the parameters of a pilot study involving one of the OEM's Simulink models (part of a powertrain[7] application). There was of course insufficient time to create any sort of analysis tool to run on the model. Instead, we agreed to apply the Salsa invariant checker to the Simulink model provided by the OEM [46], in order to see if there were any issues in the model the OEM should be aware of. The model contained a number of Simulink subsystems and Stateflow diagrams; the C code implementing the the model consisted of approximately a thousand lines. Our experience has been that models of interest to industry developers are rarely any smaller than this. In other words, any tool we wanted to develop would need to handle models with at least this much functionality.

To perform the analysis, we hand-translated the Simulink model into the SAL input notation supported by Salsa. Simulink is a large language with a complicated semantics, so this translation was the most time-consuming portion of the work. It also clarified that expecting a customer to perform such a hand-translation, or to implement an automated translator, was clearly impractical: any tool would need to natively support a language, such as Simulink, directly used by embedded software developers.

After performing this hand translation, we then applied Salsa to the generated model and discovered several anomalies in the original Simulink model, including dead code (i.e. parts of the model that would never execute) and a number of violations of the OEM's modeling policies. These results caught the attention of the OEM team, since the Simulink model had previously undergone, and passed, a thorough manual review by OEM engineers. On the other hand, some limitations of our approach were also clear. First, as mentioned above, expecting engineers to translate Simulink models into a notation like SAL was a

[7] The *powertrain* of an automobile consists of all components responsible for delivering power to a vehicle's wheels. These include the engine, driveshaft, transmission, axles, and differentials.

non-starter. Second, even if an automated translation from Simulink to SAL were implemented, we would still be constrained by the expressiveness of SAL, which supported only boolean and integer state variables and linear constraints. Based on the pilot study and subsequent discussions with the OEM team, we learned that most automotive applications, at least in the powertrain area, included both floating-point variables and non-linear constraints.

3.3 From the Concurrency Factory to Reactis

We received enthusiastic feedback from our OEM collaborators in response to the pilot study, and this became part of our application to NSF for a second, two-year round of funding. We planned to use this funding, which we were awarded in March 2001, to support the construction of the first commercial release of a tool that we could market to customers. The positive results from the pilot study led us to believe that the automotive industry would be a fertile one for our offerings, especially if our tools worked with Simulink. However, pursuing this strategy would be at odds with our initial vision of selling model-checking tools based on our own, Concurrency Factory, notation. What course should the company pursue?

Of course, from a current-day perspective, it is clear what decision we arrived at; we abandoned the Concurrency Factory and instead focused on developing testing and validation tools for Simulink, i.e. Reactis. While this strategic shift was initially driven in our minds by a market opening we perceived in a specific industry, namely, automotive, our decision also exposes important considerations regarding the commercialization of formal-verification tools. In academic formal-methods research, much work focuses on algorithms and analysis routines; less effort has traditionally been devoted to notational issues, such as modeling and specification languages. Reactive Systems' original focus on the Concurrency Factory was very much rooted in the founders' more foundational research; the expectation was that industrial users would see the benefits of formal verification and easily adapt their development processes to Reactive Systems' home-grown notations. Based on our pilot study, however, we realized that notational issues are crucially important in practice: companies want assurances that the notations they train their engineers in will be used in years to come, and they want an ecosystem of consultants and technical support to engage with on developing their design processes to accommodate new technologies. For all its intellectual rigor, the Concurrency Factory did not address these concerns. Simulink, on the other hand, had already seen significant uptake in the automotive industry, and there was substantial support for incorporating these notations into control-system design practice. Consequently, building validation and testing tools targeting Simulink removed a key barrier to developing a customer base that he Concurrency Factory would have almost certainly been unlikely to overcome.

4 Desiderata in the Design of Reactis

In order to achieve our goal of having a commercial version of Reactis ready before the second phase of our NSF funding ran out, we knew that additional technical talent would need to be added to the team. Consequently, in 2001, co-author David Hansel accepted our offer for him to join Reactive Systems.

With our technical team in place, we then needed to make a number of decisions about the design and development of Reactis. This section highlights some of the important choices that we took.

4.1 Supporting Simulink

Our experience during the pilot study with our OEM collaborators told us that supporting Simulink would be the promising route to obtaining customers in the automotive realm. The question was, how should we do this?

We decided to retain our original verification strategy, which was based on the state exploration / back-tracking approaches of on-the-fly model checking, but to re-purpose these ideas for validating Simulink diagrams. We explain the rationale for this choice more below; for now, however, we note that doing state-space exploration on Simulink diagrams of course requires the capability of generating states, computing transitions, and *backtracking*, i.e. restoring the system state to a previously constructed and stored state.

Our original hope was to use the Mathworks' simulation engine for these purposes; this would ensure that our state and transition computations would be fully consistent with those of Simulink, and also prevent us from having to implement these routines ourselves. However, the MathWorks APIs did not export the necessary functionality, and in our estimation it was quite unlikely that The MathWorks would modify these APIs to suit our purposes.

We opted instead to build our own Simulink interpreter. This approach gave us full access to all internals of states and transitions and in particular enabled us to retrieve model information necessary for state exploration with backtracking. There were significant on-going costs and challenges associated with this decision, however. To begin with, Simulink is a complex language, with a large number of operators (or "blocks") tuned for modeling control systems, and we would need to guarantee to our customers that our interpreter implemented the same semantics as The Mathworks, even as new versions of these notations were / are released. In addition, as a proprietary language, Simulink did not have a reference semantics. This lack of an independent definition of the behavior of Simulink has led to two different types of challenges over the years. The first is that corner cases in the meanings of blocks have to be uncovered via trial and error and extensive testing. The second is that customers sometimes misunderstand the intended semantics of the Simulink and will report bugs in Reactis that are, in fact, not bugs at all, but rather due to misunderstandings about the behavior of Simulink.

Our Simulink interpreter is built around a home-grown intermediate representation based on ideas from *synchronous languages* [4], of which Simulink is also an example, at least for the language's discrete (as opposed to continuous) semantics. Our simulator translates Simulink into this intermediate language; it then computes states and transitions for use in simulation on the intermediate representations. (We briefly considered using C as this intermediate notation, since there are tools for translating Simulink into C. However, these translators involve significant license fees, and for cost reasons we decided to avoid them.)

4.2 Supported Verification Approaches

Our original, Concurrency-Factory-inspired, vision featured the use of model checking to automatically determine if a given system model satisfied a formula in temporal logic. We initially considered adapting this paradigm to Simulink, but it did not fit well with our prospective customers design processes, which did not involve the formulation of such properties. However, our early discussions with industry professionals revealed interest in generic checks that should be passed by all models. For example, no model should have dead code or runtime errors such as divide-by-zero, out-of-bounds array indexing, or numerical overflows.

There was also another key role played by models in the development processes of the companies we spoke with: as specifications for the eventually deployed vehicular sub-systems (including software). For example, a model might be developed for a vehicle-transmission controller. The real-world controller hardware would contain a microprocessor on which software would be deployed to implement the behavior contained in the model. Confirming that the software source code, and the sub-system containing the deployed executable generated from the software, behaved consistently with the the model was of great interest because of the possibility of detecting and fixing errors before the sub-system was integrated into a larger vehicle design. Our industrial contacts, which had grown to included automotive OEMs and Tier-1 suppliers in the US, Europe and Japan, wondered if comprehensive suites of test cases could be automatically generated from models and used in downstream testing, with the model results being used as the oracle for judging the outcomes of software and system testing. This use of models in testing was, and is, routinely referred to as *Model-Based Testing* (MBT), and is part of a larger design paradigm called Model-Based Design (MBD) that The MathWorks markets to its customers.

We were aware of uses of model checkers for generating test cases [1, 42] in the research literature and decided to equip Reactis with support for MBT. This Reactis feature allows users to decide on a level of "thoroughness" of the test suite by specifying structural coverage criteria, such as Condition coverage, Decision coverage, and Modified Condition / Decision Coverage (MC/DC) [9] of the model. Reactis then attempts to generate test suites from the model that achieve 100% coverage of the selected criteria. While these criteria have been criticized in the literature for being insufficiently rigorous, they have the benefit of being widely understood in industry and are included in various software safety standards for automotive [28] and aerospace [44, 45].

The tool also needed support for user interaction to generate test cases, for a couple of reasons. First, users often had existing tests that they wanted incorporated into Reactis-generated test suites. Second, obtaining 100% coverage may be impossible to achieve in reasonable time, and users need to be able to manually add new tests to cover parts of the model that the Reactis test generator could not. Thus, we decided the tool needed an interactive test generator to help users with these tasks, in addition to a fully automatic test generator.

Some users also wanted to undertake model validation via testing. The general approach we decided to offer, which we call *instrumentation-based verification* [20], is based on ideas similar to the use of assertion-checking in software debugging. In traditional assertion-based debugging, users embed assertions, or boolean properties, at points in their code; at run time, these assertions are checked as the program executes, and violations are highlighted. In instrumentation-based verification, the idea is to allow users to define assertions in the form of *monitor models*, or small Simulink models with boolean outputs, that could be used as instrumentation for the main Simulink model. Generated test suites can then be run on the instrumented model, with any false outputs of monitor models flagged as errors.

Based on these considerations the earliest versions of Reactis, as well as current versions of software, included the following tools.

1. Reactis Tester generates a comprehensive yet compact test suite from a Simulink model.
2. Reactis Simulator allows users to interactively run and modify test suites on models, as well as debug models.
3. Reactis Validator checks if a Simulink model satisfies properties formulated as monitor models that have been instrumented into the model.

Reactis also uses static analysis to check for dead code, as well as a variety of run-time errors.

4.3 Verification Engine

With our desired verification functionality identified, we next needed to decide how best to implement it. In particular, the most important, and also most computationally intensive, operation was test-suite generation, and our experiences to that point led us to believe it would not be feasible to directly use existing model checkers or automatic theorem provers. The typical Simulink models that we had seen were too complex for state-of-the-art techniques then available; they often contain hundreds or even thousands of boolean, floating-point, and integer variables manipulated with the very rich (and continually growing) set of operators supported by Simulink.

Our solution to this problem was to develop (and patent) a technique, called *guided simulation* [17], for generating test cases that maximize coverage criteria. The rough idea is the following. A given coverage criterion for test suites defines a set of coverage targets that the suite must hit. For example, in Condition coverage, each atomic boolean expression in a model must be made both true and false by some test in the test suite in order for the test suite to fully cover the model. Each such expression therefore gives rise to two coverage targets: one corresponding to the expression being true, and one associated with it being false. In Reactis, a test case corresponds to a simulation run of the Simulink model being tested, with each step in the run specifying the values of the (unconnected) top-level inputs and also recording the (unconnected) top-level output values produced in response to these inputs. Thus, generating a test suite attaining 100% coverage of a given coverage criterion can be phrased as a search problem in the space of model simulation runs, with the objective of the search being to ensure that every coverage target is hit by some test case. This general search paradigm was inspired by our work with on-the-fly model-checking algorithms described in Section 2, which in effect search for proofs that a state in a model satisfies a given temporal property.

More specifically, our guided-simulation test-generation technique works by generating simulation steps, starting from an initial state of the model and tracking which coverage targets have been hit and which have not. From an intermediate state in a simulation run, Reactis selects the next inputs to provide in order to guide the next step in the simulation to the corresponding target next state, updating aggregate coverage-target data in the process. Input selection, which is a key mechanism (along with backtracking) for guiding the simulation being constructed, in turn involves two different strategies.

Constraint solving. A given coverage target can be viewed as a condition that must hold in the middle of the execution of a model. When the system is in a given state, we use techniques based on weakest preconditions to compute constraints on top-level inputs whose solutions will guarantee coverage of the given target. We then attempt to solve these constraints using *Satisfaction Modulo Theories* (SMT) technology. The initial SMT solvers included in Reactis were home-grown and supported boolean constraints and linear constraints over reals (to approximate floating-point constraints). We later switched to use the Z3 solver developed by Microsoft [36].

Monte Carlo. Values for top-level inputs are generated randomly, the target state of the simulation step is computed, and coverage information is updated appropriately.

These techniques have different strengths. When constraints can be formulated and solved, SMT-generated inputs are guaranteed to extend coverage. However, constraints may fall outside the theories supported by the constraint solver, or have solutions that are too computationally expensive to calculate. Indeed, for large models, even constructing the constraints can be infeasible. The Monte Carlo method on the other hand generates new inputs very quickly and works for any model supported by the interpreter, but at the cost of not guaranteeing growth in coverage.

Users can also place range and rate-of-change constraints on the top-level inputs of a model within Reactis. Reactis then ensures that these constraints are obeyed by any input values it generates, whether by constraint solving or randomly.

Our search strategy also uses a number of heuristics to improve test-suite coverage. These include, but are not limited to, ones for determining when a given state is a good candidate for being the source of potential future coverage and saving it; ones for when to abandon a given Monte Carlo run because coverage is not growing and backtracking to previous states that are promising; and ones for simplifying constraints to improve solver performance.

4.4 More on Coverage Metrics

The notion of coverage metrics / criteria for tests created from Simulink models derives from classical work done in the software engineering community for programming languages [37]. Some well-known metrics defined for (traditional imperative) programming languages include the following.

Statement coverage. Has every statement in a program been executed?

Decision coverage. Has every decision (i.e. maximal boolean-valued expression such as those appearing in loop guards and if-then-elese statements) in a program evaluated to both true and false?

Condition coverage. Has every atomic boolean-valued expression in a program evaluated to both true and false?

Reactis adapts these concepts to Simulink models. In addition, Reactis supports a number of metrics tailored to the Simulink notation. For example, the state-coverage criterion measures how many Stateflow states have been entered, while the transition-coverage measure tracks the number of Stateflow transitions that have fired.

The collection of metrics supported by Reactis has also expanded over time. One early evolution resulted from our work with aerospace customers. The DO-178B [44] (later updated to DO-178C [45]) safety standard for aviation software requires that the most safety-critical software components be tested to achieve 100% coverage for the modified condition/decision coverage (MC/DC) metric [9]. The ISO 26262 automotive safety standard [28] subsequently adopted MC/DC as the level of testing required for safety critical automotive software; this created significant interest in this metric among our automotive customers, and we added support for it into Reactis.

Another driver for expanding the Reactis coverage metrics was due to a desire on the part of customers to have the coverage data for Reactis-generated tests from Simulink models match the coverage results obtained from running the test cases on (automatically generated) source code. For example, in 2009 we introduced so-called *multi-block MC/DC* coverage for Simulink into Reactis. This criterion allows for the aggregation of a collection of Simulink Logical Operator blocks into a single decision for the purpose of MC/DC tracking. This criterion closely mimics the decisions created in C code that is generated from such models, and which typically involves nested boolean expressions.

5 Selling Reactis

In June of 2002, we released the first commercial version of Reactis, and the business activities of marketing and selling the tool suite moved to the forefront of our concerns. There are many aspects of building a community of customers and users around a tool, and we will not describe all of these here. However, some features of building and maintaining a successful commercial tool have impacts on decisions regarding its technical development, and we describe several of these in this section.

Engineers in industry, especially those in a company's *business units* (as opposed to research and development), have numerous responsibilities related to a company's source of revenue — its products and services. For this reason, they typically have limited time to explore new tools and think about how they might evolve their development processes to accommodate them. In order to successfully sell Reactis, we needed to develop materials, including white papers, presentations and other marketing collateral, that explained not only the functionality of Reactis, but also how it could be deployed in different industrial scenarios. A substantial amount of our post-release resources were devoted to developing these materials, and to visiting prospective customers both to market our tool but also to learn how their internal model-based design processes were organized, so that we could offer tailored suggestions on how Reactis could be usefully deployed. We also offered training courses for customers at their sites.

In our experience, driving adoption of Reactis within companies has required extensive collaboration between Reactive Systems and the Reactis users within those companies. New users will need training, and users of all types will from time to time need advice on how to access and best use tool functionality. Users will also stumble across functionality they don't expect (either intended, or buggy), and want guidance from Reactive Systems on what to do. For its part, Reactive Systems gains direct insights into user experiences through its interactions with users, which give valuable feedback for refining Reactis to flatten the learning curve and best address specific needs in user-development processes. For this reason, the company has policies for responding quickly to all user requests. These interactions consumes significant resources, but we view user support as a key component of our business strategy.

Tool users also often request new tool features. A case in point for Reactis was its treatment of C code embedded in Simulink diagrams. Simulink offers several ways to incorporate C code into a model, including S-Functions, C Caller blocks, and C code called from Stateflow. In each case, a portion of the model's functionality is determined by this C code. Early versions of Reactis treated C code in a model as a collection of black boxes. Reactis would compile the embedded C code and use it to compute the semantics of the given model component, thereby enabling simulation and test generation for these models. However, coverage was not tracked in the C code, and it was not possible to step into the C code for debugging (another use case of the Reactis Simulator). In response to user requests for the capability of tracking coverage information for this C code, in 2007 Reactive Systems launched the Reactis for C Plugin product, which enabled white-box testing for the C code parts of a model. With the C Plugin, coverage targets are identified within the C code, and Reactis Simulator is able to step into the C code during simulation. After positive feedback from customers on the C Plugin, in 2011 the company packaged this functionality for C into a separate tool suite, Reactis for C, which supports Reactis validation capabilities, including test generation, coverage tracking, and assertion verification, on ANSI C code. This tool reused significant parts of the original Reactis verification engine and drew upon experience we gained in developing customized versions of the Concurrency Workbench of North Carolina using the Process Algebra Compiler [18].

Of course, promising new functionality but not delivering it until much later can sour tool users on the tool in question. In the case of Reactis, we addressed this by scheduling new releases twice a year and also making beta versions of new functionality we were planning on releasing to users who had requested it.

The importance of engagement with, and responsiveness to, users cannot be overstated, and it highlights a crucial difference between traditional research tools and commercial ones: significant technical resources must be devoted to ensuring users can effectively and efficiently use tool functionality in order for the tools to be successful in the marketplace. In an academic setting, it can be difficult to justify these expenditures, since researchers and students naturally wish to focus on implementing new ideas of their own. In an industrial setting, on the other hand, without users there cannot be a tool!

6 Current State of Reactis

The most recent version, V2023.2, of Reactis was released in December of 2023. This release includes new versions of the following tools.

- Reactis Tester, Reactis Simulator and Reactis Validator, which target test-generation, model debugging and model validation for Simulink.
- Reactis for C Plugin, which is used in conjunction with the Reactis tools for Simulink to provide analytical capabilities C code that may be embedded in models.
- Reactis for C, which provides testing, simulation and testing capabilities for ANSI C code.

Each of the above categories of tools are licensed separately. That is, users would buy separate licenses to Reactis, which includes Tester, Simulator and Validator; to Reactis for C Plugin (which requires a license for Reactis); or to a license to Reactis for C (which is independent of the other tools).

In addition, the V2023.2 release includes licensing possibilities for two additional tools. The Reactis for EML Plugin, first released in 2015, supports white-box testing of the Embedded MATLAB (EML) portions of a model. EML is the subset of the MATLAB language that MathWorks supports for the generation of embedded C code. Simulink offers the capability to incorporate EML into a Simulink model, for example, in a MATLAB Function block or as a callable from Stateflow. With the EML Plugin, coverage targets are identified within the EML code, Reactis Tester attempts to exercise these targets, and Reactis Simulator steps into EML code for debugging and coverage visualization. To use the Reactis for EML Plugin, a user must also have a license for Reactis. Our Reactis Model Inspector tool enables users to navigate models. It does not include testing or validation functionality and is intended to support (read-only) model auditing and review.

7 Conclusions and Perspectives on the Future

In this paper we have given a brief history of the development of Reactis, a commercial tool suite we have developed over the past 20 years that uses ideas from formal methods to provide testing and validation support for Simulink and ANSI C. In the course of the paper, we have highlighted both the foundations of Reactis in model-checking research but also places where we needed to depart from the academic tool-development paradigm, and why. We are very strong believers in the value of formal methods. We also believe that adapting the techniques to industrial settings requires sustained effort and insight into company needs and concerns, so that engineers in these organizations can derive the benefits of this technology and verification-tool developers can maximize the impact of their offerings.

In terms of future plans for Reactis, at present, companies in over 20 different countries have acquired Reactis. Most users are in the automotive industry, although a significant number of users work in aerospace and defense companies. In general, different industries have different modeling cultures and use different notations as a result. While Simulink users are widespread, other modeling tools, such as LabVIEW [38], have strong user bases as well. Developing a "Reactis generator" for such modeling notations is something we see as a possibly fruitful direction for future development, as a way of expanding the Reactis market. Of course, such a development would need a business case, to include not only a significant body of users but also an industrial culture that is willing to pay for tools that support software design, development, and validation. We also are tracking developments in SMT tools with a view towards enhancing future versions of Reactis.

In terms of future prospects for industrial uptake of formal verification, we see reasons for optimism, provided the tools are appropriately targeted. For example, we do not see much of a market for "generic software-verification tools" that would be on par with, for example, compilers in terms of their widespread adoption. A primary reason for our pessimism on this front is two-fold. On the one hand, generic software-development tools are widely expected to be no-cost in the marketplace, which limits the financial return available to developers wishing to introduce formal-methods capabilities into them. On the other hand, much "generic software" has traditionally had relatively relaxed expectations in terms of functional correctness on the part of the software's users, meaning software companies have incentives to devote more development resources to new features and less to quality assurance. However, in many “non-software” industries whose products include significant software, expectations on the part of users, regulatory agencies, and insurance companies for correct functional behavior are much higher, and we see ongoing significant opportunities for formal methods in these sectors. The automotive and aerospace / defense sectors fall into this category; so do medical devices, household appliances, infrastructure (electricity, water, transportation, data communications, etc.), and banking, to name a few. Finally, consumers, companies, and governments are increasingly concerned about privacy and cybersecurity. While historically there may have been a certain tolerance for software functionality shortcomings, there is growing push-back against cybersecurity vulnerabilities. A 2024 report [39] by the US Office of the National Cybersecurity Director highlights the importance of formal methods as a tool for improving cybersecurity, and the cloud-services sector, for example, offers an interesting future opportunity for commercial applications of formal methods for this purpose, in our view.

References

1. Ammann, P., Black, P., Majurski, W.: Using model checking to generate tests from specifications. In: Second International Conference on Formal Engineering Methods. pp. 46–54 (1998). https://doi.org/10.1109/ICFEM.1998.730569
2. Baier, C., Katoen, J.P.: Principles of model checking. MIT Press (2008)
3. Barrett, C., Conway, C., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23. pp. 171–177. Springer (2011)
4. Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P., De Simone, R.: The synchronous languages 12 years later. Proceedings of the IEEE **91**(1), 64–83 (2003)
5. Bergstra, J.A., Ponse, A., Smolka, S.A.: Handbook of process algebra. Elsevier (2001)
6. Bharadwaj, R., Sims, S.: Salsa: Combining constraint solvers with BDDs for automatic invariant checking. In: 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 1785, pp. 378–395. Springer (2000)
7. Bhat, G., Cleaveland, R., Grumberg, O.: Efficient on-the-fly model checking for CTL. In: 10th Annual IEEE Symposium on Logic in Computer Science. pp. 388–397. IEEE (1995)
8. Bhat, G., Cleaveland, R.: Efficient local model-checking for fragments of the modal μ-calculus. In: International Workshop on Tools and Algorithms for the Construction and Analysis of Systems. pp. 107–126. Springer (1996)
9. Chilenski, J., Miller, S.: Applicability of modified condition/decision coverage to software testing. Software Engineering Journal **9**(5), 193–200 (1994)
10. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 93–107. Springer (2013)
11. Clarke, E., Emerson, E., Sistla, A.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems (TOPLAS) **8**(2), 244–263 (1986)

12. Cleaveland, R.: Tableau-based model checking in the propositional mu-calculus. Acta Informatica **27**, 725–747 (1990)
13. Cleaveland, R., Madelaine, E., Sims, S.: A front-end generator for verification tools. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS 95). Lecture Notes in Computer Science, vol. 1019, pp. 153–173. Springer (1995). https://doi.org/10.1007/3-540-60630-0_8
14. Cleaveland, R., Gada, J.N., Lewis, P.M., Smolka, S.A., Sokolsky, O., Zhang, S.: The Concurrency Factory – Practical tools for specification, stimulation, verification, and implementation of concurrent systems. Specification of Parallel Algorithms **18**, 75–90 (1994)
15. Cleaveland, R., Hennessy, M.: Testing equivalence as a bisimulation equivalence. Formal Aspects of Computing **5**, 1–20 (1993)
16. Cleaveland, R., Parrow, J., Steffen, B.: The Concurrency Workbench: A semantics-based tool for the verification of concurrent systems. ACM Transactions on Programming Languages and Systems (TOPLAS) **15**(1), 36–72 (1993)
17. Cleaveland, R., Sims, S.T., Hansel, D.: System and method for automatic test-case generation for software (Jan 5 2010), US Patent 7,644,398
18. Cleaveland, R., Sims, S.T.: Generic tools for verifying concurrent systems. Science of Computer Programming **42**(1), 39–47 (2002), special Issue on Engineering Automation for Computer Based Systems
19. Cleaveland, R., Smolka, S.A., Lewis, P.M., Ramakrishna, Y.: Specification and verification for concurrent systems with graphical and textual editors (May 7 2002), US Patent 6,385,765
20. Cleaveland, R., Smolka, S.A., Sims, S.T.: An instrumentation-based approach to controller model validation. In: Model-Driven Development of Reliable Automotive Services: Second Automotive Software Workshop, ASWSD 2006, San Diego, CA, USA, March 15-17, 2006, Revised Selected Papers 2. pp. 84–97. Springer (2008)
21. Cleaveland, R., Steffen, B.: A linear-time model-checking algorithm for the alternation-free modal mu-calculus. Formal Methods in System Design **2**, 121–147 (1993)
22. Du, X., Smolka, S.A., Cleaveland, R.: Local model checking and protocol analysis. International Journal on Software Tools for Technology Transfer **2**, 219–241 (1999)
23. Hansel, D., Cleaveland, R., Smolka, S.A.: Distributed prototyping from validated specifications. Journal of Systems and Software **70**(3), 275–298 (2004)
24. Heitmeyer, C., Jeffords, R., Labaw, B.: Automated consistency checking of requirements specifications. ACM Trans. Softw. Eng. Methodol. **5**(3), 231–261 (Jul 1996)
25. Henzinger, M., Henzinger, T., Kopke, P.: Computing simulations on finite and infinite graphs. In: IEEE 36th Annual Foundations of Computer Science. pp. 453–462. IEEE (1995)
26. Holzmann, G.: On-the-fly model checking. ACM Computing Surveys (CSUR) **28**(4es), 120–es (1996)
27. Holzmann, G.: The model checker SPIN. IEEE Transactions on Software Engineering **23**(5), 279–295 (1997)
28. ISO 26262: Road vehicles – Functional safety (2011)
29. Kanellakis, P., Smolka, S.: CCS expressions, finite state processes, and three problems of equivalence. In: Second Annual ACM Symposium on Principles of Distributed Computing. pp. 228–240 (1983)
30. Kanellakis, P., Smolka, S.: CCS expressions, finite state processes, and three problems of equivalence. Information and Computation **86**(1), 43–68 (1990)

31. Kozen, D.: Results on the propositional μ-calculus. Theoretical Computer Science **27**(3), 333–354 (1983). https://doi.org/10.1016/0304-3975(82)90125-6, special Issue Ninth International Colloquium on Automata, Languages and Programming (ICALP) Aarhus, Summer 1982
32. Liu, X., Ramakrishnan, C., Smolka, S.: Fully local and efficient evaluation of alternating fixed points. In: Tools and Algorithms for the Construction and Analysis of Systems: 4th International Conference, Lisbon, Portugal, March 28–April 4, 1998 Proceedings 4. pp. 5–19. Springer (1998)
33. Liu, X., Smolka, S.: Simple linear-time algorithms for minimal fixed points. In: Automata, Languages and Programming: 25th International Colloquium, ICALP'98 Aalborg, Denmark, July 13–17, 1998 Proceedings 25. pp. 53–66. Springer (1998)
34. Malhotra, J., Smolka, S., Giacalone, A., Shapiro, R.: Winston: A tool for hierarchical design and simulation of concurrent systems. In: Rattray, C. (ed.) Specification and Verification of Concurrent Systems. pp. 140–152. Springer London, London (1990). https://doi.org/10.1007/978-1-4471-3534-0_7
35. Milner, R.: Communication and concurrency. Prentice Hall, Englewood Cliffs, New Jersey (1989)
36. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
37. Myers, G., Badgett, T., Thomas, T., Sandler, C.: The art of software testing. John Wiley & Sons (2004)
38. National Instruments, https://www.ni.com/en/shop/labview.html
39. Office of the National Cybersecurity Director: Back to the building blocks: A path towards secure and measurable software. The While House (26 February 2024)
40. Parnas, D.: Tabular representation of relations. Tech. Rep. 260, Communications Research Laboratory, McMaster University (October 1992)
41. Ramakrishna, Y., Ramakrishnan, C., Ramakrishnan, I., Smolka, S., Swift, T., Warren, D.S.: Efficient model checking using tabled resolution. In: Computer Aided Verification: 9th International Conference, CAV'97 Haifa, Israel, June 22–25, 1997 Proceedings 9. pp. 143–154. Springer (1997)
42. Rayadurgam, S., Heimdahl, M.: Coverage based test-case generation using model checkers. In: Eighth Annual IEEE International Conference and Workshop On the Engineering of Computer-Based Systems–ECBS 2001. pp. 83–91 (2001). https://doi.org/10.1109/ECBS.2001.922409
43. Reactive Systems, Inc., https://reactive-systems.com/
44. RTCA DO-178B (EUROCAE ED-12B), "Software considerations in airborne systems and equipment certification," 2nd edition (1992)
45. RTCA DO-178C, "Software considerations in airborne systems and equipment certification" (2011)
46. Sims, S., Cleaveland, R., Butts, K., Ranville, S.: Automated validation of software models. In: 16th Annual International Conference on Automated Software Engineering (ASE 2001). pp. 91–96. IEEE (2001)
47. Sokolsky, O., Smolka, S.: Incremental model checking in the modal mu-calculus. In: Computer Aided Verification: 6th International Conference. pp. 351–363. Springer (1994)
48. Stirling, C., Walker, D.: Local model checking in the modal mu-calculus. Theoretical Computer Science **89**(1), 161–177 (1991)
49. Zhang, S., Sokolsky, O., Smolka, S.: On the parallel complexity of model checking in the modal mu-calculus. In: Ninth Annual IEEE Symposium on Logic in Computer Science. pp. 154–163. IEEE (1994)