Noname manuscript No.
(will be inserted by the editor)

Supporting Incremental Behaviour Model Elaboration

Sebastian Uchitel[†⋆] · Dalal Alrajeh[†] · Shoham Ben-David[◦] · Victor Braberman[⋆] · Marsha Chechik[◦] · Guido De Caso[⋆] · Nicolas D'Ippolito[†] · Dario Fischbein[†] · Diego Garbervetsky[⋆] · Jeff Kramer[†] · Alessandra Russo[†] · German Sibay [†]

Received: date / Accepted: date

Abstract Behaviour model construction remains a difficult and labour intensive task which hinders the adoption of model-based methods by practitioners. We believe one reason for this is the mismatch between traditional approaches and current software development process best practices which include iterative development, adoption of use-case and scenario-based techniques and viewpoint- or stakeholder-based analysis; practices which require modelling and analysis in the presence of partial information about system behaviour.

Our objective is to address the limitations of behaviour modelling and analysis by shifting the focus from traditional behaviour models and verification techniques that require full behaviour information to partial behaviour models and analysis techniques, that drive model elaboration rather than asserting adequacy. We aim to develop sound theory, techniques and tools that facilitate the construction of partial behaviour models through model synthesis, enable partial behaviour model analysis and provide feedback that prompts incremental elaboration of partial models.

In this paper we present how the different research threads that we have and currently are developing help pursue this vision as part of the "Partial Behaviour Modelling - Foundations for Interative Model Based Software Engineering" Starting Grant funded by the ERC. We cover partial behaviour modelling theory and construction, controller synthesis, automated diagnosis and refinement, and behaviour validation.

We thank the ERC for financially supporting this work through the grant StG PBM-FIMBSE

[†] Imperial College London, UK.
E-mail: [s.uchitel,d.alrajeh,n.dippolito,d.fischbein,g.sibay,a.russo]@imperial.ac.uk
· [†] FCEN, Universidad de Buenos Aires, Argentina.
E-mail: [suchitel, gcaso, vbraber, diegog]@dc.uba.ar
· [⋆] University of Toronto, Canada.
E-mail: chechik@cs.toronto.edu,shohambd@gmail.com

Keywords Partial Behaviour Modelling

1 Introduction

Software systems are amenable to analysis through the construction of behaviour models. This corresponds to the traditional engineering approach to construction of complex systems. Models can be studied to increase confidence on the adequacy of the product to be built. The advantage of using behaviour models to describe systems is that they are cheaper to develop than the actual system. Consequently, they can be analysed and mechanically checked for properties in order to detect design errors early in the development process and allow cheaper fixes.

To address this problem significant effort has been devoted to developing approaches for modelling and verifying system behaviour. These approaches are typically supported by automated tools that allow specifying behaviour models, using them as prototypes for exploring the system behaviour, and checking adequacy of the model to properties and system requirements.

Although behaviour modelling and analysis has been shown to be successful in uncovering subtle requirements and design errors, adoption by practitioners has been slow. Partly, this is due to the complexity of building behavioural models in the first place – behaviour modelling remains a difficult, labour-intensive task that requires considerable expertise. In addition, and perhaps more importantly, the benefits of the analysis appear only at the end of a costly process of constructing a comprehensive behaviour model.

The reason for the latter is that most approaches to behaviour modelling require a complete description of the system behaviour for a fixed scope and span [47]: the specification is assumed to completely describe the system with respect a chosen set of phenomena types (scope) that determine the level of abstraction of the model and a chosen part of the problem or solution domain (span). The completeness assumption is limiting in the context of software development process best practices which include iterative development, adoption of use-case and scenario-based techniques and viewpoint- or stakeholder-based analysis; practices which require modelling and analysis in the presence of partial information about system behaviour.

The limitations described above motivate a series of research questions that we aim to address: How can the construction of behaviour models be significantly simplified? Can we provide automated or semi-automated procedures to assist engineers in building initial approximations of system behaviour? Can we provide feedback early in the model construction effort, even in the presence of partial behaviour descriptions? Can this feedback be used to prompt further model elaboration?

Our objective is to address the limitations of behaviour modelling and analysis by shifting the focus from traditional behaviour models and verification techniques to partial behaviour models and analysis techniques that drive model elaboration rather than asserting adequacy. The vision we advo-

cate does not require complete descriptions for analysis to proceed. We aim to provide a framework in which useful feedback can be obtained even when very little information regarding system behaviour is available.

In the remainder of this paper we summarise the different research threads we have been working on, namely partial behaviour models (Section 2), controller synthesis (Section 3), automated diagnosis and refinement (Section 4), and behaviour validation (Section 5).

2 Partial Behaviour Models

Labelled transition systems (LTS) are the basis for widely used techniques for modelling and analysing the behaviour of software systems. An LTS is a transition system where transitions are labelled with actions. The set of actions of an LTS is called its communicating alphabet and constitutes the interactions that the modelled system can have with its environment. Behaviour models for complex systems are built compositionally by describing the behaviour of each system component with an LTS and constructing a composite LTS model that exhibits the emergent behaviour of the components executing asynchronously while synchronizing on shared actions.

Existing semantics for LTS, and other popular behaviour modelling formalisms such as statecharts, tend to assume that the model provides a complete description with respect to its alphabet. For instance, in a trace-based semantics, the traces described explicitly by the transitions of the model are assumed to describe all the intended executions of the system. Any trace not reproducible through the transitions is assumed to be undesired behaviour of the system. This interpretation is consistent with semantics based on standard equivalence relations such as strong and weak bisimulation [59].

An alternative interpretation of LTS is that they represent an upper or lower bound to the acceptable behaviour of the system. Consider, for instance, behaviour models synthesised automatically from scenario-based specifications such as message sequence charts [46]. Scenarios provide examples of how system components, the environment and users work concurrently and interact in order to provide system level functionality. Example-based specifications such as scenarios are naturally partial as it is impractical and often infeasible to provide a comprehensive description on an example by example basis. Consequently, when behaviour model synthesis is applied to a scenario description, the resulting model represents a lower bound on the intended system behaviour: the synthesised model describes some of the behaviour required in the final implementation, but the fact that the model does not exhibit a particular behaviour does not mean that the intended implementation must not provide it. Elaboration of an LTS that represents a lower bound on intended system behaviour consists in strictly adding new behaviour (possibly described as new scenarios) to the model while preserving the hitherto known behaviour.

The interpretation of LTS models as an upper bound to system behaviour corresponds to the classical view of process-oriented specification [44,59] where an LTS is interpreted as a specification of all the acceptable behaviour of the system. In this view, implementations that satisfy the specification must provide, in some sense, a subset of the behaviour described in the LTS. The notions of trace and failures refinement [44], or that of simulation [59] are classic formalisations of this interpretation.

The three interpretations discussed above (complete, upper, and lower bounds) are not restricted to LTS alone. They apply to the many other traditional operational formalisms for describing system behaviour such as those with richer notions of state (e.g kripke structures[52]) and transitions (e.g. interface automata [23]). Yet these interpretations, particularly the more widespread one which assumes completeness, are limiting.

For instance, traditional behaviour models cannot adequately model the known behaviour of a system that has been described by a combination of use-cases, scenarios and safety properties, a plausible situation in current development practices. This is because the use- cases and scenarios provide a lower-bound and the safety properties provide an upper bound to the intended system behaviour. It is clear that the interpretations for LTS described above cannot capture the required behaviour from the scenarios, the proscribed behaviour from the properties and the rest which has not been explicitly proscribed nor required. This is because traditional behaviour models are essentially two-valued: the transitions model all of the required behaviour and the rest is proscribed, the transitions model some of the required behaviour and the rest is possible, or the transitions model all of the possible behaviour and the rest is proscribed.

In fact, and more generally, the problem arises because traditional behaviour models cannot model explicitly what is unknown about system behaviour, or, in other words, distinguish between the behaviour that the system is must provide, from what it must not provide, and from what is yet unknown. Behaviour models that distinguish between these kinds of behaviour are referred to as partial behaviour models. A number of such models exist, and promising results on their use to support incremental modelling and viewpoint analysis have been reported (e.g., Partial Labelled Transition Systems [72]), Modal Transition Systems (MTS)[54], Mixed Transition Systems[20] and multi-valued Kripke structures [17]).

Partial behaviour models, such as Modal Transition Systems (MTS) [54], distinguish between three kinds of behaviour: required, proscribed and unknown, and therefore can describe *both* an upper and a lower bound to the intended system behaviour, allowing both bounds to be refined simultaneously. For instance, MTS are equipped with two kinds of transitions *required* transitions and *possible* transitions. The former provide a upper bound to system behaviour, while the latter provide the lower bound to system behaviour.

The semantics of a partial behaviour model can be thought of as a set of traditional behaviour models. For instance, MTS semantics can be given in terms of sets of LTS that provide all of the behaviour required by the MTS, do

not provide any of the behaviour proscribed by the MTS, and make arbitrary decisions on the MTS's unknown behaviour. Intuitively, as more information becomes available, unknown or unclassified behaviour gets changed into either required or proscribed behaviour. The notion of refinement between MTS captures this intuition formally and provides an elegant way of describing the process of behaviour model elaboration as one in which behaviour information is acquired and introduced into the behaviour model incrementally, gradually refining an MTS until it characterizes a single LTS.

The original notion of refinement was aimed at comparing MTS models with the same alphabet and no unobservable transitions and is referred to as strong refinement [54]. Although in [54] a notion of weak refinement that allows for unobservable actions was defined, we have extended this notion to account for models that have different alphabets [69,35]. More recently [34, 33], we presented a an alternative, possibly more appropriate observational refinement, based on branching equivalence [74].

A particularly useful notion in the context of software and requirements engineering is that of *merge*. Merging two consistent models is a process that should result in a minimal common refinement of both models where *consistency* is defined as the existence of one common refinement. Intuitively, merging builds a model that characterises the intersection of the LTS characterised by the models being merged. In other words, the merge characterises the LTS that provide all the required behaviour of the MTS being merged, and that do not provide any of the proscribed behaviour of the MTS being merged.

MTS merging can be used as the conjunction of multiple partial operational descriptions. The original formulation of merge was done by Larsen in [55] where an incomplete merge algorithm was proposed for MTS under strong refinement. Recently we have presented a correct and complete algorithm [36]. The problem of merge under observational refinements is still open, a partial result can be found in [35] where we present an incomplete algorithm for merging models with different alphabets under weak refinement. Finally, we have studied the problem of providing feedback when MTS are inconsistent and cannot be merged [66].

MTS, with any of its refinement semantics, has the limitation of not being closed under merge. This has been studied for strong refinement extensively and various extensions that resolve this problem have been proposed, notably Disjunctive MTS [53]. However, this is not the case when weak semantics is used: infinite state DMTS are required. We are currently studying variants of MTS which are closed for such weak refinements.

We have revisited the problem of behaviour model synthesis in the context of MTS. We have provided a generic extension of synthesis approaches that start from existential scenario-based specifications and build LTS models [70]. Furhtermore, in [71] we also show how MTS can support synthesis from heterogenous specifications including safety properties and existential scenarios.

Given that MTS are more expressive than LTS, a common target for scenario synthesis, we have explored opportunities for defining novel synthesis approaches that start from more expressive scenarios notations. In particular,

we have investigated synthesis from triggered scenarios (both with existential and universal modalities [67]). Existential triggered scenarios had hitherto been neglected in existing scenario description languages (e.g., [42]) as it is impossible to adequately capture their semantics using traditional behaviour models: They express branching properties on required and possible behaviour (when the trigger occurs, a branch satisfying the main chart must exist but other behaviour may be allowed). This in turn has led to experimentation in logics which have sufficient expressive power to describe existential triggered scenarios without requiring full branching capability [12].

We have developed a testbed for manipulating MTS models, including synthesis, analysis, merge, parallel composition and animation in a tool, the Modal Transition System Analyser [30], which is currently an open source program available at `http://sourceforge.net/projects/mtsa/`.

3 Behaviour Model Synthesis

Michael Jackson's Machine-World model [48] establishes a framework on which to approach the challenges of requirements engineering. In this model, requirements R are prescriptive statements of the world expressed in terms of phenomena on the interface between the machine we are to build and the world in which the real problems to be solved live. Such problems are to be captured with prescriptive statements expressed in terms of phenomena in the world (but not necessarily part of the machine-world interface) called goals G and descriptive statements of what we assume to be true in the world (domain assumptions D).

Within this setting, a key task in requirements engineering is to understand and document the goals and the characteristics of the domain in which these are to be achieved, in order to formulate a set of requirements for the machine to be built such that assuming that the domain description and goals are valid, the requirements in such domain entail the goals, i.e., $R, D \models G$.

Thus, a key problem of requirements engineering can be formulated as a synthesis problem. Given a set of descriptive assumptions on the environment behaviour and a set of system goals, construct an operational model of the machine such that when composed with the environment, the goals are achieved. Such problem is known as the controller synthesis problem [64, 62] and has been studied extensively. Controller synthesis algorithms have been used in various software engineering settings including synthesis of glue code and component adaptors in order to achieve safe composition at the architecture level [10], and particularly in service oriented architectures [15], or to synthesise adaptation strategies in autonomous systems [68].

We have investigated the use of controller synthesis techniques to aid the incremental elaboration of behaviour models. Focus has been on adapting and extending existing controller synthesis in order to gain insight on given models of system goals G and domain assumptions D while attempting to synthesise an operational model for the requirements R such that $R, D \models G$. In other

words, the point is not so much to build R, but rather to investigate how the non-existence of an R such that $R, D \models G$ can prompt the elaboration of both G and D.

3.1 Environment Assumptions

Jackson [48] and others (e.g., [77,75,60]) have argued that environment assumptions play a key role in the requirements validation process. Many system failures are due to invalid assumptions, many times related to an over-idealisation of the environment's behaviour. In other words, statements regarding environment behaviour that are not realistic are used to demonstrate the correctness of the requirements with respect to the goals. However, given that the assumptions are invalid, when the system is developed and deployed, the goals are not achieved. Thus, best practices include explicit modelling of assumptions not only better support validation but also make explicit when system goals are guaranteed to be achieved, helping to set more realistic expectations.

Although assumptions and their relation with the synthesis problem has been studied recently [18,16], most synthesis approaches (e.g., [14,68,43]) do not support an explicit distinction between assumptions and goals. On the other hand, techniques that do support explicit specification of assumptions such as [61] give assumptions a syntactic treatment; no semantic restriction or methodological guidelines are provided as to what assertions consitute valid assumptions. This is crucial, as we show in [28], when assertions proposed as domain assumptions are not *realisable* [75] by the environment, then controller synthesis techniques can produce valid, yet useless, results: A controller which rather than attempting to achieve the specified goals, succeeds in violating the system assumptions and thus discharging its obligation to fulfill the goals.

In [28] we present a controller sythesis technique and methodological guidelines for synthesising event-based behaviour models. The approach works for an expressive subset of liveness properties, GR(1) [61], that distinguishes between controlled and monitored actions, and differentiates between system goals and environment assumptions. The technique adapts and extends recent advances [61] in controller synthesis for shared memory communication style.

In [29] we further this work by studying failure assumptions of environment controlled actions and identify a realistic fairness condition on failures, *strong independent fairness*, which allows for a polynomial treatment of the control sythesis problem. Intuitively, the strong independent fairness condition states that not only every failure and every assumption must occur fairly (infinitely often if enabled infinitely often) but also independently of system state and of every other failure and assumptions. In other words, failures and assumptions cannot be coordinated. They must be "controlled" by different agents which must be oblivious of each other. This notion of fairness has a corresponding interpretation in stochastic behaviour: if the environment can be thought of as a grounding of a probabilistic environment with non-zero probability of

non-failure, then the executions that are not strong independent fair have probabilistic measure zero.

Applying controller synthesis techniques which require explicit description of assumptions prompts behaviour model elaboration. For instance, as part of the validation of the work in [29] we studied the controller synthesised by [13] for a web-service required to coordinate purchases on a furniture-sales service and deliveries on a shipping service. The case study includes failures such as furniture-sales and shipping services responding negatively to requests. The technique in [13] gives no guarantees on the resulting controller satisfiying the expected goals. Hence the resulting controller is not guaranteed to satisfy purchase requests it receives. Applying our technique [29] shows that the assumptions for this system are insuficient to construct a guaranteed controller and that in fact progress and fairness conditions are required. If these conditions are explicitly added to the behaviour specification, then a controller that guarantees system goals is possible and is constructed using [29].

Thus the lack of initial realisability of the specification of the case study in [13] led to a more elaborate description of the domain assumptions required to guarantee system level goals.

3.2 Partial Environment Models

Existing controller synthesis approaches require complete descriptions of the problem domain. Typically, the domain is described in a formal language with its semantics defined as some variation of a two-valued state machine such as Labelled Transition Systems (LTS) [49] or Kripke structures [52]. Thus, the model of the problem domain is assumed to be complete up to some level of abstraction (i.e., with respect to an alphabet of actions or propositions).

As discussed previously, traditional behaviour modelling frameworks based on LTS and Kripke structures are not well suited for describing partial knowledge about the problem domain. However, controller synthesis techniques for partial behaviour modelling formalisms such as multi-valued Kripke structures [37]) and Modal Transition Systems (MTS) [54] has yet to be studied.

In [27], we define controller synthesis in the context of partially specified problem domains. More specifically, we study the problem of checking the existence of an LTS controller (i.e., controller realisability) capable of guaranteeing a given goal when deployed in a completely defined LTS domain model that conforms to a partially defined problem domain given as an MTS.

More specifically, given that an MTS defines a set of LTS implementations, we define the *MTS control problem* as responding if all, none or some of the LTS implementations an MTS describes admit an LTS controller that guarantees a given goal expressed as a Fluent Linear Temporal Logic (FLTL) [39] formula.

A technique that yields an answer to the MTS control problem is presented in [27] showing that, despite dealing with a potentially infinite number of LTS implementations, the MTS control problem is in the same complexity class as the underlying LTS synthesis problem. Furthermore, the results for

MTS realisability can be used with controller synthesis techniques that deal efficiently with restricted yet expressive goals such as [9,8,61].

We believe that the feedback resulting from addressing the MTS control problem can prompt partial model elaboration. This is particularly so when the answer to the realisability question is "some". In these cases, a refinement of the partial behaviour model that prunes out the implementations which cannot be controlled is necessary, representing an oportunity for elicitation.

4 Automated Diagnosis and Repair

At the heart of model elaboration is the model-analyse-elaborate cylce. Engineers produce models, be the partial or complete with respect to a particular level of abstraction, and then use automated sound tools to analyse the emergent behaviour of these models. The result of the analysis can be positive, in which the engineer considers that sufficient confidence on the adequacy of the model has been achieved, or negative, in which concrete examples of undesired behaviour are identified. The elaboration phase attempts to revise or refine the existing model to eliminate undesired behaviour while preserving the desired behaviour exhibited by the model.

A key family of tools in the model-analyse-elaborate cycle is that of model-checkers. Model Checking is an automated technique for verifying formal artefacts. It has been successfully used to verify system design and requirements in different domains including communication and security protocols as well as biological systems. A model checker requires a model provided in some formal language and a semantic property that such model is expected to have. Such property is described in a formal language (possibly different from the one used for the model, but with compatible semantics). The model checker then automatically checks the validity of the specified property in the models semantics [19]. If the property is found to not hold, a counterexample is generated which shows how the property can be falsified.

The automatic generation of counterexamples is one of model checking's powerful features for system fault detection. Counter-examples are meant to help engineers in the tasks of identifying the cause of a property violation and correcting the model. However, these tasks are complex and little automated support exists for them. Even in relatively small models such tasks are far from trivial since (i) counterexamples are expressed in terms of the model's semantics rather than the language used to describe the model or the property, (ii) counterexamples show the symptom and do not indicate the cause of the violation and (iii) any manual modification to the model may fail to resolve the problem and may even introduce violations to other desirable properties.

Inductive logic programming (ILP) is a subfield of machine learning which uses logic programming as a uniform representation of knowledge to perform explanation reasoning. Given an encoding of the known background knowledge and a set of positive and negative examples, an ILP system will compute a

hypothesis which, in conjunction with the background knowledge, allows all the positive but none of the negative examples.

In the context of behaviour model elaboration, ILP can be applied by representing the model under analysis as the background knowledge, property violations produced by a model checker as negative examples and any witness to the property (these can easily be generated by a model checker) deemed to be valid by the engineer as positive examples. The computed hypothesis is a statement which refines or revises the model specification and that is guaranteed to avoid the property violations while preserving the property witnesses.

We believe that that model checking and ILP can be seen as two complementary approaches which if integrated appropriately can support model elaboration in general and behaviour model elaboration in particular. We have successfully applied the combination of model checking and ILP in software engineering settings to tackle a variety of problems related to behaviour model elaboration. We succintly discuss each one in the remainder of the section. Note, however, that although each of these problem domains differ in their modelling language, semantics and class of elaborations, they share a number of characteristics. Their theories describe event-based systems, with a model semantics expressed in terms of finite-state transition systems, whose ontology features and semantic properties can naturally be captured by Event Calculus logic programs [50] with stable model semantics [38].

4.1 Learning Operational Requirements from Goals

A key activity in requirements engineering is the elaboration and analysis of operational requirements. Operational requirements are requirements for each operation that is to be provided by the software. Such requirements can, and typically are, described using pre-, post- and trigger-conditions.

Little support exists for the elaboration of operational requirements from high-level goals. Letier and van Lamwsveerde [56] have developed an approach based on operationalisation patterns which allows the derivation of operational requirements in the form of pre- and trigger-conditions from goals expressed in Linear Temporal Logic (LTL). Requirements generated by this approach are guaranteed to be correct. However, patterns are restricted to a collection of goal and requirement templates, and their application requires a fully refined goal model. Consequently, the elaboration of operational requirements from goals remains constrained to the set of templates and can be labour intensive and error-prone. The availability of a more systematic and automated approach would therefore benefit the process of operationalising goals.

In [2,6], we present a formal, tool-supported framework that combines model checking and ILP to elaborate operational requirements, in the form of pre- and trigger-conditions, that are *correct* and *complete* with respect to a given set of system goals. The framework is defined as an iterative process that consists of four conceptual phases. First, in the *analysis* phase, an existing partial specification of operational requirements is verified against a

given goal model using a model checker. If verification is unsuccessful (i.e., the operational requirements do not entail the goals), the example of goal violantion automatically generated is used in the *scenario elaboration* phase where an engineer elaborates it into a set of positive and negative scenarios. In the *learning phase*, the partial specification of operational requirements and scenarios are used by a non-monotonic inductive learning system to compute a set of operational requirements that covers all positive scenarios and eliminates all negative ones. Finally, in the *selection* phase, the engineer selects the operational requirements to be added from the list proposed by the learning phase. The four phases are then repeated until no goal violation is detected.

The approach was validated by using case studies [51,32]. For each of the systems studied, we had an informal description of the system-to-be, a linear temporal logic representation of its high level goals and a formal operationalisation of the goals, that is, a set of operational requirements that is complete with respect to the goals. It is important to note that all these elements, informal description, goals, and operational requirements, were produced by a third-party. The validation consisted in starting from the high-level goals and applying the iterative method described in [2]. Human interventions required by the approach (for example and counter-example generation) were performed based on our understanding of the informal description of the system and the high-level goals. Having completed all iterations, the operational requirements learned were compared to the ones provided. In all cases, we were able to learn the provided operational requirements, however, in some cases we were also able to identify alternative operationalisations of the high-level goals.

4.2 Zeno-Behaviour Elimination

In requirements engineering [48] focus is on prescriptive declarative statements of intent whose satisfaction requires the cooperation of agents (or active components) in the software and its environment. Such statements are commonly referred to as system-level goals, or simply goals [76].

The declarative nature of goals often hinders the application of a number of successful validation techniques based on executable models such as graphical animations, simulations, and rapid-prototyping. They do not naturally support narrative style elicitation techniques, such as those in scenario-based requirements engineering and are not suitable for down-stream analyses that focus on design and implementation issues which are of an operational nature.

To address these limitations, techniques have been developed for constructing behaviour models automatically from declarative descriptions in general [71] and from goal models specifically [57]. The core of these techniques is based on temporal logic to automata transformations developed in the model checking community. For instance, in [57] Labelled Transition Systems (LTS) are built automatically from KAOS goals expressed in fluent linear temporal logic [39].

A key technical difficulty in constructing behaviour models from goal models is that the latter are typically expressed in a synchronous, non-interleaving semantic framework while the former have an asynchronous interleaving semantics. This mismatch relates to the fact that it is appropriate to make different assumptions for modelling requirements and system goals than for modelling communicating sequential processes. One of the practical consequences of this mismatch is that the construction of behaviour models from a goal model may introduce deadlocks and progress violations. More specifically, the resulting behaviour model may be *zeno*, i.e exhibit traces in which time never progresses. Clearly, these models do not adequately describe the intended system behaviour and thus are not a suitable basis for analysis.

A solution proposed in [57] to the problem of zeno traces is to construct behavior models from a fully *operationalised* goal model rather than from a set of high-level goals. This involves identifying system operations and extracting operational requirements in the form of *pre-* and *trigger-conditions* from the high-level goals [56]. This has some important disadvantages. Firstly, operationalisation is a manual process for which only partial support for the derivation of a complete operationalised model is provided. Support comes in the form of derivation patterns restricted to some common goal patterns [22]. Secondly, it impedes early construction of behaviour models from high-level goals which can provide insights before going through a tedious operationalisation process.

In [7,4] we apply a combination of model checking and ILP to the problem of non-zeno behaviour model construction. The approach starts with a goal model and produces a non-zeno behaviour model that satisfies all goals. Briefly, the proposed method first involves translating automatically the goal model, formalised in Linear Temporal Logic (LTL), into a (potentially zeno) labelled transition system. Then, in an iterative process, zeno traces in the behaviour model are identified mechanically, elaborated into positive and negative scenarios, and used to automatically learn preconditions that prevent the traces from occurring. Identification of zeno traces is achieved by model checking the behaviour model against a time progress property expressed in LTL, while preconditions are learned using Inductive Logic Programming (ILP).

As a result of the proposed approach, not only a non-zeno behaviour model is constructed, but also a set of precondition is produced. These preconditions, in conjunction with the known goals, ensure the non-zeno behaviour of the system. Consequently, the approach also supports the operationalisation process of goal models described in [56].

4.3 Obstacle Generation

Completeness is among the most critical and difficult challenges facing requirements engineers. Missing requirements and assumptions are reported as one of the major causes of software failure [76]. Incompleteness often arises from the lack of anticipation of exceptional conditions. The natural inclination is

rather to conceive idealised systems; this prevents adverse events or conditions from being properly identified, and as a result, specifications of suitable countermeasures in such circumstances are missing.

Risk analysis is therefore at the heart of the requirements engineering process [31,76]. A *risk* is commonly defined as an uncertain factor whose occurrence may result in some loss of satisfaction of some corresponding objective. In goal-oriented system modelling frameworks, obstacles are introduced as a natural abstraction for risk analysis when using goal models [77]. An *obstacle* to a goal is a precondition for the non-satisfaction of this goal. Depending on the category of goal being obstructed, obstacles may correspond to safety hazards, security threats, inaccuracy conditions on software input/output variables with respect to their environment counterpart, etc.

Obstacle analysis roughly consists of three steps [76]: (a) identify as many obstacles as possible to every leaf goal in the systems goal refinement graph, (b) assess the likelihood and severity of each obstacle; and (c) resolve likely and severe obstacles by systematic transformations to the goal model using appropriate countermeasures.

The obstacle identification step is obviously crucial. In [77], a formal technique is described for generating obstacles by regressing goal negations through available domain properties. Although quite systematic, this technique appears costly to implement for goals formalised in a first-order real-time linear temporal logic. No tool support is available.

In [3], we present an alternative, tool-supported technique for obstacle generation. A complete set of obstacles, relative to what is known about the domain, is computed by iterating the following cycle: (a) a behaviour model is synthesised from the available background properties; (b) this model is verified against the goal and against a negated form of it, in order to generate a negative trace (counterexample) and a positive trace (witness), respectively; (c) the negative trace is taken as a positive example whereas the positive trace is taken as a negative example input for a learning engine; (d) the learning tool generates a set of candidate obstacles that cover the positive example and exclude the negative one; (e) the user can then select from the generated obstacles those considered likely and severe, and suggest further domain properties; (f) a new cycle is applied to the background properties augmented with such properties and the negated obstacles generated at the previous cycle. The process terminates when a domain-complete set of obstacles is generated for the available domain properties.

4.4 Vacuity Resolution for Triggered Scenarios

Scenarios, use cases and story boards are popular means for supporting requirements engineering activities. They illustrate examples of how the software-to-be and its environment should and should not interact. They are commonly used as an intuitive, semi-formal language for describing behaviour at a functional level.

A common form for providing examples of behaviour is through conditional statements. Use cases [1] support existential conditional statements such as "once an appropriate user ID and passwords has been obtained, a homeowner *can* access the surveillance cameras placed throughout the house from any remote location via the internet" [63]. Live Sequence Charts [42] support universal conditional statements such as "the controller should probe the thermometer for a temperature value every 100 milliseconds, and if the result is more that 60 degrees, it *should* deactivate the heater and send a warning to the console". Some languages support both existential and universal conditional scenarios [67].

Conditional scenarios with different modalities are useful. They provide support for "what-if" elaboration of requirements specifications [1], and the progressive shift from existential statements, in the form of examples and use-cases, to universal statements in the form of declarative properties. Each conditional scenario constitutes only a partial description of the system's intended behaviour. Hence, typically many of them are used in conjunction along with other behaviour descriptions such as system goals [21]. The emergent behaviour of such rich descriptions can be complex to reason about, hindering validation, and resulting frequently in specifications that are incomplete or contradictory.

One particular issue that conditional scenarios have is that they are liable to being satisfied vacuously; a system can be constructed so that it satisfies the conditional scenarios by never satisfying the condition. For instance, a system in which the homeowner is never given a user password vacuously satisfies the use case described above. This problem, commonly referred to as *antecedent failure* [11] in temporal specifications, is often an indication that the specification is partial and hence provides an opportunity for elicitation; it is clear that the stakeholder's intention is that "the system should provide the user with an id and password", and if it does, then the user can access the installed surveillance cameras. In addition, vacuously satisfiable specifications can have pernicious effects, concealing conflicting behaviour which is important to explore. For example, consider two scenarios extracted from the mine pump example in [51]: "once the methane sensors detect that the methane level is critical, *then* the pump controller must send a signal to the pump to be switched off" and "once the water sensors detect that the water level is above the high-threshold, *then* the pump controller must send a signal to the pump to be switched on". These scenarios are consistent as a system in which water sensors never detect high water and methane levels vacuously satisfies both scenarios. However, if these two levels were to occur, then the scenarios provide contradictory information of what the controller must do.

In [5] we present an approach that not only detects vacuously satisfiable conditional scenarios but also provides automated support for learning new scenarios that ensure the conditions, i.e. triggers, are satisfied. More specifically, the approach takes as input a set of scenarios formalised as triggered existential and universal scenarios [67] and consists of two main phases. The first involves (i) synthesising a Modal Transition System from the scenarios,

representing all possible implementations that satisfy them and (ii) performing a vacuity check, using a model checker, against a scenario's trigger. If the vacuity check is positive, the model checker produces examples of how the system-to-be could satisfy the trigger, i.e. non-vacuity witnesses [41]. In the second phase, (iii) an engineer classifies the examples as either positive or negative, i.e. ones that should be accepted or not in the final implementation, and then (iv), together with the given scenarios, inputs them into an inductive logic programming learning tool to compute new triggered scenarios which, if added to the existing scenarios, guarantee that they are no longer vacuously satisfiable. This process is repeated for each given triggered scenario, producing in the end a scenario-based specification that is not vacuously satisfiable.

5 Behaviour Model Validation

Behaviour model validation aims to determine the degree to which a behaviour model is a sufficiently accurate representation of the real world (as it is or as it is intended to be once the system under construction is deployed). The gap from the formal language used for modelling to the untractable informal world makes validation a difficult task.

Although related, verification of behaviour models (determining whether the behaviour model satisfies specific formally described properties [45]) is of a very different nature, where at least the artefacts to be compared are in the realm of mathematics.

Behaviour model verification and validation are complementary activities; both are necessary to increase confidence regarding the quality of the software under construction. Much work has gone into supporting behaviour model verification; however, we believe, there is significant progress to be made on supporting behaviour model validation.

There are two broad strategies that can be taken to validate behaviour models. One is to turn the validation problem into a verification one. More concretely, to produce a specification against which the behaviour can be verified. The idea is that if the specification is simpler than the artefact, validation of the former is likely to be simpler and less error prone. Although an effective strategy, since an alternative specification is required, it must be validated appropriately, falling back into the validation problem. In other words, turning a validation problem into a verification problem creates a new (possibly simpler but of reduced scope) validation task, so eventually human intervention is required.

The other strategies require a human in the loop that constrasts informally the model against his or her understanding of the domain. Walkthroughs, inspections and reviews are classic structured activities for organising this task. However, key to effective validation is the ability to present behaviour models in alternative, semantic preserving, views. Hence, much work has gone into developing semantic preserving automated manipulations of models that can be used to produce alternative views. Some classic examples of this strategy

are minimisation, slicing, execution, simulation and abstraction. We have been pursuing the latter for a number of years. More specifically, we have focussed on automated abstraction for validation of pre/post condition specifications and API implementations with *requires* clauses.

5.1 Behaviour Validation of Pre/Post Condition Specifications

Pre- and post-condition specifications constitute good practice in a variety of behaviour modelling activities. In requirements engineering, they provide the link between declarative high-level system goals and operational requirements for the software-to-be [73]. Use case specifications, which are popular in development processes such as RUP (Rational Unified Process), are also equipped with pre- and post-conditions. In design, the notion of design by contract [58], as a mechanism to abstract the way functionality is provided by a procedure or method, is underpinned by pre-/post- conditions. Object oriented design commonly includes design of method pre- and post-conditions in addition to the specification of class or object invariants. At the code level, the use of assertions to verify at run-time pre-/post- conditions is considered good practice [65].

A pre/post condition pair constitutes a specification that is local to a specific operation (method, procedure, use case, event, etc.). The precondition is an assertion that is expected to hold before the occurrence of the operation. The postcondition is an assertion that is guaranteed to hold after the occurrence of the operation if the precondition held before the occurrence. Typically, a contract specification will include various operations (needed to provide some significant service) each with a pre/post condition pair and possibly an invariant that is expected to hold after the occurrence of any sequence of the specified operations.

Validating pre/post condition specifications, i.e., understanding if there is a correspondence between the meaning of the specification and the meaning that the specification was expected to have, is a difficult and error prone task. Although understanding the pre/post condition for a single operation may be relatively simple, understanding if chaining them for an arbitrary sequence of operations is describing the intended outcome is complicated. For example, ensuring that the pre/post conditions of a set of operations of an API are correct requires understanding if they preserve the system invariant and fit together adequately to provide the intended API functionality. Validating a use case model requires understanding how the various use-cases can be combined to provide the expected software-wide requirements.

In [25], we propose a strategy for validation of pre/post condition specifications based on the conjecture that pre/post condition specifications would benefit from easily auditable abstractions that exhibit global implications of locally specified behaviour. The approach is based on the static construction of a conservative *abstraction* of the specification semantics (typically, an infinite

state machine) in the form of a finite behaviour model that is sufficiently small to make validation tractable.

The technique buids abstractions at a level which can be seen as a generalisation of the pre/post condition philosophy: A precondition describes the state in which a specific operation is permissible. We are interested in capturing the precondition for each arbitrary set of operations. In other words, each state in the resulting behaviour model should characterise the condition for which a subset of the specified operations is enabled; this means that the invariant of the state is the conjunction of the preconditions enabled at that state. The contract abstraction is then completed by adding transitions according to the preconditions and postconditions of the operations they model: A transition can be added if the precondition for the operation holds on the source state and the postcondition holds on the target state.

The models constructed by the approach described herein can be used to validate contract pre/post-condition-based specifications through inspection, animation and simulation. We believe, and our experience so far confirms, that the criterion chosen for abstraction facilitates validation and debugging. Firstly, because a formal and intuitive correspondence exists between the state space of the behaviour model and that of the artifact being specified. Furthermore, that correspondence is structured in a way that can be easily traced back to the original specification. Not only does each state in the behaviour model represent an invariant expressed in terms of the variables, predicates and propositions that appear in the specification (and hence constructing concrete scenarios from abstract ones is straightforward), but also the invariants are expressed as a conjunction of preconditions, each of which is a building block of the specification being validated (and hence facilitating the identification of problematic operations). Secondly, in the case studies conducted so far, the state-based models we have produced automatically from contract specifications have had a similar level of abstraction to models manually produced by the authors of the contract specification. For instance, we have produced abstractions that correspond to manually produced typestate specifications for object oriented classes [26], and abstractions that are comparable to the state-machines included in Microsoft technical documents to aid the comprehension of their protocol specifications.

5.2 Program Behaviour Validation

Code artefacts that have non-trivial requirements with respect to the order in which their methods or procedures ought to be called are commonplace. Such is the case for many API implementations and objects. In practice, descriptions of intended behaviour are incomplete and informal, if documented at all, hindering verification and validation of the code artefacts themselves and the client code that uses the artefacts.

The work in [24] addresses the problem of validating if API implementations provide their intended behaviour when descriptions of this behaviour are

informal, partial or non-existent. Validation of API implementation behaviour can result in the identification of bugs in the code which induce undesired requirements, adjustment of the requirements expected by the engineer to the requirements implicit in the code, and the improvement of available documentation for that code.

Although seemingly a technique that is applicable further downstream than validation of behaviour models, this is not necesarily the case. It is not uncommon to find in industry behaviour models specified with a standard programming languages (albeit in a restricted form). For instance, in the model-based testing approach used by the Protocol Engineering Team at Microsoft [40] behaviour models are provided as C# classes. Each class has methods that are interpreted as guarded rules defining a rich action machine. In [24] we report on the application of this technique to Microsoft protocols, among others.

The technique described in [24] and then extended [78] automatically constructs abstractions based on enabledness equivalence from code artefacts equipped with requires clauses for methods. These models, similarly to typestates, encode all admissible sequences of method calls. The level of abstraction at which such models are constructed aims at preserving enabledness of sets of operations, resulting in a finite model with intuitive semantics and formal traceability links to the code.

Evaluation of this work shows that enabledness-based abstractions can be useful for validation of code artefacts and identifying findings that relate to bugs in code and problems in expected or documented requirements. Evaluation was performed on case studies such as the `PipedOutputStream`, `Signature` and `ListItr` from the Java Development Kit (JDK) 1.4 implementation; the `SMTPProtocol` class from the RISTRETTO protocol-level Java mail client; and the `PCCRR` class was taken from a C# SpecExplorer protocol model. Resulting abstractions were reviewed by an expert and compared to informal third-party developed documentation that was already available for these clasess. These reviews led to the identification of issues related to mismatches between code and documentation.

6 Conclusions

We have presented the main threads of research that we have and currently are developing in the context of the "Partial Behaviour Modelling - Foundations for Interative Model Based Software Engineering" Starting Grant funded by the ERC. The project is concerned with supporting incremental elaboration of behaviour models by providing feedback early in the modelling effort and by prompting issues that can drive further model elaboration.

Thus, the vision we are pursuing is one in which complete descriptions are not needed before analysis and feedback are possible. Rather, we aim to provide a framework in which useful feedback can be obtained even when very little information regarding system behaviour is available.

We have described the four main threads of research that pursue this vision. The first, partial behaviour models, aims at studying behaviour models capable of explicitly describing behaviour that is yet to be elicited, yet to be defined or simply uneconomical to describe at a certain stage. We have made contributions in this thread related to Modal Transition System refinement, merge, synthesis, and analysis. The second thread, Controller Synthesis, aims at developing controller synthesis techniques to automatically build models of system requirements such that they guarantee system goals under specific environment assumptions, but also to investigate how the non-existence of such a controller can prompt elaboration of system goals and assumptions. We have made contributions adapting controller synthesis techniques to a software engineering setting based on event-based behaviour models while focusing on methodologically sound use of assumptions in such techniques. The third thread, automated diagnosis and refinement, aims at combining model checking and inductive logic programming to support elaboration by providing suggestions, sound by construction, of refinements or revisions of partial or inconsistent models. We have contributed a framework for combining model checking and inductive logic programming in addition to various applications of this framework to model elaboration. Finally, the last thread has focused on the problem of validation of behaviour to support human inspection. We have contributed a novel abstraction technique that has shown to aid the identification of issues in behaviour specifications and code.

Although for presentation purposes our work has been structured in four main threads, in practice these threads overlap and play on each other. For instance, the non-existence of a suitable controller for all concrete environments described by a partial behaviour model leads naturally to refine the model pruning out uncontrollable environments. Achieving such pruning can be done using automated learning. Furthermore, once a controller has been synthesised, validating its behaviour can be done by producing enabledness-based abstractions. The synergetic use of the breadth of contributions described in this paper conforms much of the future work needed to further our vision of incremental, iterative elaboration of partial behaviour models.

References

1. I. Alexander and N. Maiden. *Scenarios, stories, use cases: through the systems development life-cycle*. Wiley, 2004.
2. D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel. Learning operational requirements from goal models. In *Proc. of 31st Intl. Conf. on Softw. Eng.*, pages 265–275, 2009.
3. D. Alrajeh, J. Kramer, A. van Lamsweerde, A. Russo, and S. Uchitel. Generating obstacle conditions for requirements completeness. In *Proc. of 34th Intl. Conf. on Softw. Eng.*, 2012.
4. Dalal Alrajeh, Jeff Kramer, Alessandra Russo, and Sebastián Uchitel. Deriving non-zeno behaviour models from goal models using ilp. *Formal Asp. Comput.*, 22(3-4):217–241, 2010.
5. Dalal Alrajeh, Jeff Kramer, Alessandra Russo, and Sebastián Uchitel. Learning from vacuously satisfiable scenario-based specifications. In Juan de Lara and Andrea Zisman,

editors, *FASE*, volume 7212 of *Lecture Notes in Computer Science*, pages 377–393. Springer, 2012.
6. Dalal Alrajeh, Oliver Ray, Alessandra Russo, and Sebastián Uchitel. Using abduction and induction for operational requirements elaboration. *J. Applied Logic*, 7(3):275–288, 2009.
7. Dalal Alrajeh, Alessandra Russo, and Sebastián Uchitel. Deriving non-zeno behavior models from goal models using ilp. In José Luiz Fiadeiro and Paola Inverardi, editors, *FASE*, volume 4961 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2008.
8. R. Alur and S. La Torre. Deterministic generators and games for LTL fragments. *ACM Transactions on Computational Logic (TOCL)*, 5(1):1–25, 2004.
9. E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller synthesis for timed automata. In *Proceedings of the IFAC Symposium on System Structure and Control*, 1998.
10. M. Autili, P. Inverardi, M. Tivoli, and D. Garlan. Synthesis of" correct" adaptors for protocol enhancement in component-based systems. *SAVCBS 2004 Specification and Verification of Component-Based Systems*, page 79, 2004.
11. D. Beatty and R. Bryant. "Formally Verifying a Microprocessor Using a Simulation Methodogoly". In *Proceedings of Design Automation Conference'94*, pages 596–602, 1994.
12. Shoham Ben-David, Marsha Chechik, Arie Gurfinkel, and Sebastián Uchitel. Cssl: a logic for specifying conditional scenarios. In Tibor Gyimóthy and Andreas Zeller, editors, *SIGSOFT FSE*, pages 37–47. ACM, 2011. (Acceptance rate: 16%. Scopus).
13. P. Bertoli and M. Pistore. Planning with extended goals and partial observability. In *Proceedings of ICAPS*, volume 4, 2004.
14. Piergiorgio Bertoli, Alessandro Cimatti, Marco Pistore, Marco Roveri, and Paolo Traverso. MBP: a model based planner. In *Proceedings of the IJCAI01 Workshop on Planning under Uncertainty and Incomplete Information*, 2001.
15. A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli. Automatic synthesis of behavior protocols for composable web-services. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium*, pages 141–150. ACM, 2009.
16. Krishnendu Chatterjee, Thomas A. Henzinger, and Barbara Jobstmann. Environment assumptions for synthesis. In *Proceedings of the 19th international conference on Concurrency Theory*, CONCUR '08, pages 147–161, Berlin, Heidelberg, 2008. Springer-Verlag.
17. Marsha Chechik, Benet Devereux, Steve Easterbrook, and Arie Gurfinkel. Multi-valued symbolic model-checking. *ACM Trans. Softw. Eng. Methodol.*, 12(4):371–408, October 2003.
18. Marsha Chechik, Mihaela Gheorghiu, and Arie Gurfinkel. Finding environment guarantees. In *Proceedings of the 10th international conference on Fundamental approaches to software engineering*, FASE'07, pages 352–367, Berlin, Heidelberg, 2007. Springer-Verlag.
19. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
20. D. Dams, R. Gerth, and O. Grumberg. "Abstract Interpretation of Reactive Systems". *ACM Transactions on Programming Languages and Systems*, 2(19):253–291, 1997.
21. A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1):3–50, 1993.
22. R. Darimont and A. van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. In *Proc. of 4th ACM SIGSOFT symposium on Foundations of Softw. Eng.*, pages 179–190, 1996.
23. Luca de Alfaro and Thomas A. Henzinger. Interface automata. *SIGSOFT Softw. Eng. Notes*, 26(5):109–120, September 2001.
24. Guido de Caso, Víctor A. Braberman, Diego Garbervetsky, and Sebastián Uchitel. Program abstractions for behaviour validation. In Richard N. Taylor, Harald Gall, and Nenad Medvidovic, editors, *ICSE*, pages 381–390. ACM, 2011.
25. Guido de Caso, Víctor A. Braberman, Diego Garbervetsky, and Sebastián Uchitel. Automated abstractions for contract validation. *IEEE Trans. Software Eng.*, 38(1):141–162, 2012.

26. R. DeLine and M. Fahndrich. Typestates for Objects. *Ecoop 2004-Object-Oriented Programming: 18th European Conference, Oslo, Norway, June, 2004: Proceedings*, 2004.
27. Nicolás D'Ippolito, Victor Braberman, Nir Piterman, and Sebastián Uchitel. "The Modal Transition System Control Problem". In *Submitted*.
28. Nicolás D'Ippolito, Victor Braberman, Nir Piterman, and Sebastián Uchitel. Synthesising non-anomalous event-based controllers for liveness goals. *ACM Trans. Softw. Eng. Methodol.*, 22(1), 2013.
29. Nicolás D'Ippolito, Víctor A. Braberman, Nir Piterman, and Sebastián Uchitel. Synthesis of live behaviour models for fallible domains. In Richard N. Taylor, Harald Gall, and Nenad Medvidovic, editors, *ICSE*, pages 211–220. ACM, 2011.
30. Nicolás D'Ippolito, Dario Fischbein, Marsha Chechik, and Sebastián Uchitel. Mtsa: The modal transition system analyser. In *ASE*, pages 475–476. IEEE, 2008.
31. M.S. Feather and S.L. Cornford. Quantitative risk-based requirements reasoning. *J. Requirements Eng.*, 8:248–265, 2003.
32. A. Finkelstein. The london ambulance system case study. In *Proc. of 8th Intl. Work. on Software Specification and Design*, pages 5–19, 1996.
33. D. Fischbein. *"Foundations for Behavioural Model Elaboration Using Modal Transition Systems "*. PhD thesis, Imperial College London, UK, April 2012.
34. Dario Fischbein, Víctor A. Braberman, and Sebastián Uchitel. A sound observational semantics for modal transition systems. In Martin Leucker and Carroll Morgan, editors, *ICTAC*, volume 5684 of *Lecture Notes in Computer Science*, pages 215–230. Springer, 2009.
35. Dario Fischbein, Nicolás D'Ippolito, Greg Brunet, Marsha Chechik, and Sebastián Uchitel. Weak alphabet merging of partial behavior models. *ACM Trans. Softw. Eng. Methodol.*, 21(2):9, 2012.
36. Dario Fischbein and Sebastián Uchitel. On correct and complete strong merging of partial behaviour models. In Mary Jean Harrold and Gail C. Murphy, editors, *SIGSOFT FSE*, pages 297–307. ACM, 2008.
37. Melvin Fitting. "Many-Valued Modal Logics". *Fundamenta Informaticae*, 15(3-4):335–350, 1991.
38. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R.A. Kowalski and K. Bowen, editors, *Proc. of 5th Int. Conference on Logic Programming*, pages 1070–1080, 1988.
39. D. Giannakopoulou and J. Magee. "Fluent Model Checking for Event-Based Systems". In *Proceedings of the 9th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'03)*, pages 257–266. ACM Press, September 2003.
40. W. Grieskamp, N. Kicillof, K. Stobie, and V. Braberman. Model-based quality assurance of protocol documentation: tools and methodology. *STVR*, (in press).
41. A. Gurfinkel and M. Chechik. "How Vacuous Is Vacuous?". In *Proceedings of 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *LNCS*, pages 451–466, Barcelona, Spain, March 2004. Springer.
42. David Harel. *Come, let's play - scenario-based programming using LSCs and the play-engine*. Springer, 2003.
43. William Heaven, Daniel Sykes, Jeff Magee, and Jeff Kramer. Software engineering for self-adaptive systems. chapter A Case Study in Goal-Driven Architectural Adaptation, pages 109–127. Springer-Verlag, Berlin, Heidelberg, 2009.
44. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, New York, 1985.
45. IEEE. IEEE Standard Glossary of Software Engineering Terminology, September 1990.
46. ITU. Message sequence charts. Technical Report Recommendation Z.120, International Telecommunications Union. Telecommunication Standardisation Sector, 2000.
47. Michael Jackson. *Software Requirements & Specifications - A Lexicon of Practice, Principles and Prejudices*. Addison-Wesley, 1995.
48. Michael Jackson. The world and the machine. In *Proceedings of the 17th international conference on Software engineering*, ICSE '95, pages 283–292, New York, NY, USA, 1995. ACM.
49. R. Keller. "Formal Verification of Parallel Programs". *Communications of the ACM*, 19(7):371–384, 1976.

50. R.A. Kowalski and M. Sergot. A logic-based calculus of events. *New generation computing*, 4(1):67–95, 1986.
51. J. Kramer, J. Magee, and M. Sloman. Conic: An integrated approach to distributed computer control systems. In *IEE Proc., Part E 130*, 1983.
52. S.A. Kripke. "Semantical Considerations on Modal Logic". *Acta Philosophica Fennica*, 16:83–94, 1963.
53. K. Larsen and L. Xinxin. "Equation Solving Using Modal Transition Systems". In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science (LICS'90)*, pages 108–117. IEEE Computer Society Press, 1990.
54. K.G. Larsen and B. Thomsen. "A Modal Process Logic". In *Proceedings of 3rd Annual Symposium on Logic in Computer Science (LICS'88)*, pages 203–210. IEEE Computer Society Press, 1988.
55. Kim G. Larsen, Bernhard Steffen, and Carsten Weise. "A Constraint Oriented Proof Methodology based on Modal Transition Systems". In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'95)*, LNCS, pages 13–28. Springer, May 1995.
56. E. Letier and A. Van Lamsweerde. Deriving operational software specifications from system goals. In *Proc. of 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 119–128, 2002.
57. Emmanuel Letier, Jeff Kramer, Jeff Magee, and Sebastián Uchitel. Deriving event-based transition systems from goal-oriented requirements models. *Autom. Softw. Eng.*, 15(2):175–206, 2008.
58. B. Meyer. Applying 'design by contract'. *Computer*, 25:40–51, 1992.
59. R. Milner. *Communication and Concurrency*. Prentice-Hall, New York, 1989.
60. David Lorge Parnas and Jan Madey. "Functional Documents for Computer Systems". *Science of Computer Programming*, 25:41–61, 1995.
61. Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive (1) designs. *Lecture notes in computer science*, 3855:364–380, 2006.
62. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 179–190. ACM New York, NY, USA, 1989.
63. R.S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 7th edition, 2010.
64. PJG Ramadge and WM Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.
65. David S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, 1995.
66. Mathieu Sassolas, Marsha Chechik, and Sebastián Uchitel. Exploring inconsistencies between modal transition systems. *Software and System Modeling*, 10(1):117–142, 2011.
67. German Sibay, Sebastián Uchitel, and Víctor A. Braberman. Existential live sequence charts revisited. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *ICSE*, pages 41–50. ACM, 2008.
68. Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. Plan-directed architectural change for autonomous systems. In Arnd Poetzsch-Heffter, editor, *SAVCBS*, pages 15–21. ACM, 2007.
69. S. Uchitel and M. Chechik. "Merging Partial Behavioural Models". In *Proceedings of 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 43–52, November 2004.
70. Sebastián Uchitel, Greg Brunet, and Marsha Chechik. Behaviour model synthesis from properties and scenarios. In *ICSE*, pages 34–43. IEEE Computer Society, 2007.
71. Sebastián Uchitel, Greg Brunet, and Marsha Chechik. Synthesis of partial behavior models from properties and scenarios. *IEEE Trans. Software Eng.*, 35(3):384–406, 2009.
72. Sebastián Uchitel, Jeff Kramer, and Jeff Magee. Behaviour model elaboration using partial labelled transition systems. In *ESEC / SIGSOFT FSE*, pages 19–27. ACM, 2003.
73. H.T. Van, A. van Lamsweerde, P. Massonet, and C. Ponsard. Goal-oriented requirements animation. In *Requirements Engineering Conference, 2004.*, pages 218–228, 2004.
74. Rob J. van Gabbeek and W. Peter Weijland. Branching time and abstraction in bisimulation semantics. *J. ACM*, 43(3):555–600, 1996.

75. A. Van Lamsweerde. Goal-Oriented Requirements Engineering: A Guided Tour. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*. IEEE Computer Society Washington, DC, USA, 2001.
76. A. van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.
77. Axel van Lamsweerde and Emmanuel Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Transactions on Software Engineering*, 26:978–1005, October 2000.
78. Edgardo Zoppi, Víctor Braberman, Guido de Caso, Diego Garbervetsky, and Sebastián Uchitel. Contractor.net: inferring typestate properties to enrich code contracts. In *Proceedings of the 1st Workshop on Developing Tools as Plug-ins*, TOPI '11, pages 44–47, New York, NY, USA, 2011. ACM.