Improving Reliability of Cooperative Concurrent Systems with Exception Flow Analysis

Fernando Castor Filho[1], Alexander Romanovsky[2], Cecília Mary F. Rubira[3]

[1] *Informatics Center, Federal University of Pernambuco Av. Prof. Lus Freire s/n, 50740-540, Recife, PE, Brazil*

[2] *School of Computing Science, Newcastle University Newcastle, UK. NE1 7RU*

[3] *Institute of Computing, State University of Campinas P.O. Box 6176, 13084-971 Campinas, SP, Brazil*

Abstract

Developers of fault-tolerant distributed systems need to guarantee that fault tolerance mechanisms they build are in themselves reliable. Otherwise, these mechanisms might in the end negatively affect overall system dependability, thus defeating the purpose of introducing fault tolerance into the system. To achieve the desired levels of reliability, mechanisms for detecting and handling errors should be developed rigorously or formally. We present an approach to modeling and verifying fault-tolerant distributed systems that use exception handling as the main fault tolerance mechanism. In the proposed approach, a formal model is employed to specify the structure of a system in terms of cooperating participants that handle exceptions in a coordinated manner, and coordinated atomic actions serve as representatives of mechanisms for exception handling in concurrent systems. We validate the approach through two case studies: (i) a system responsible for managing a production cell, and (ii) a medical control system. In both systems, the proposed approach has helped us to uncover design faults in the form of implicit assumptions and omissions in the original specifications.

Email addresses: `fcastor@acm.org` (Fernando Castor Filho), `alexander.romanovsky@newcastle.ac.uk` (Alexander Romanovsky), `cmrubira@ic.unicamp.br` (Cecília Mary F. Rubira).

1 Introduction

Applications that could potentially endanger human lives or lead to great financial losses are usually made fault-tolerant [1] so that they are capable of providing their intended service, even if only partially, when errors occur. Fault-tolerant systems include mechanisms for detecting errors in their states and recovering from them. There are two main types of error recovery [1]: backward and forward. The former is based on rolling a system back to its previous correct state and generally uses either diversely implemented software or simple retry; the latter involves transforming the system into any correct state, is typically application-specific and relies on an exception handling mechanism [2,3].

Usually, a significant part of the system code is devoted to error detection and handling [2,4]. In 1989, Cristian [2] claimed that, for telephone switching applications, this often amounted to more than two thirds of the overall system code. A more recent study [4] of a set of open-source applications written in Java discovered that between 1 and 5% of the program text consisted of exception handlers (`catch` blocks) and clean-up actions (`finally` blocks). In another study [5], focusing on five large-scale applications based on the Java Enterprise Edition [6] platform, the ratio of the number of exception handlers to that of operations in each application varied between 0,058 and 1,79. Finally, some of us have conducted yet another study [7], involving four applications. Two of themwere produced in industry and two in academia. In this case, the ratio of the number of handlers to that of operations ranged from between 0,099 to 0,208.

In spite of the pervasiveness of error detection and handling code, it is usually the least understood, tested or documented [2] in a system. This is mainly due to the tendency among developers to focus on the normal activity of applications and only deal with the code responsible for error detection and handling at the implementation phase. What is more, there are other issues that aggravate this situation in distributed systems, such as the high cost of reaching an agreement, the lack of a global view on the system state, multiple concurrent errors, difficulties in ensuring error isolation, etc. All of these factors complicate the development of reliable systems in general and of mechanisms that make them reliable in particular. The overall result is that the parts of a system responsible for making it reliable are usually the source of design faults [2,5,4].

For the desired levels of reliability to be achieved in a system, error detecting and handling mechanisms should be systematically applied from the early phases of development [8]. Moreover, the construction of these fault tolerance mechanisms should follow a rigorous or formal development methodology [9]. In this manner, these mechanisms are made more reliable and do not introduce new faults into the system.

1.1 Problem

The concept of Coordinated Atomic (CA) Actions [10] was developed by combining distributed transactions and atomic actions. The latter are used to control cooperative concurrency and to implement exception handling [11], whereas the former [12] are used to maintain the consistency of resources shared by competing actions. CA actions function as exception-handling contexts for cooperative systems, and exceptions raised in an action are handled cooperatively by all of its participants. If two or more exceptions are concurrently raised, an *exception resolution mechanism* [11] is employed to identify an exception that represents all the exceptions raised concurrently (a *resolved exception*) in order to handle it. Many case studies [13–16] have shown that CA actions are a powerful and useful tool for structuring large distributed fault-tolerant systems. In this paper, we view CA actions as representative of mechanisms for exception handling in distributed systems.

In order for CA actions to be applicable in constructing complex real-world systems with strict dependability requirements, software development based on CA actions needs to be supported with rigorous models, techniques, and tools. Several approaches have been proposed to formalize the CA action concept aiming to either offer a more complete and rigorous description of the concept [17] or to verify CA action-based designs [15]. However, there is an important aspect of CA actions that has not been properly addressed by existing work, and that is coordinated exception handling. This is surprising, since exception handling complements other techniques in improving reliability, such as atomic transactions, and promotes the implementation of specialized and sophisticated error recovery measures. Moreover, in some distributed applications, a rollback is not possible or is prohibitively expensive. In this scenario, exception handling may be the only sensible choice available.

Some authors [18] claim that mechanisms for involving multiple participants in order to cooperatively handle exceptions are difficult for both implementation and use. We believe, however, that programmers will make more mistakes in an ad hoc implementation of cooperative exception handling than in applying well-defined mechanisms provided by such general frameworks as CA actions. There is thus a need for techniques and tools that would mitigate the inherent complexity of exception handling in a concurrent setting and help developers in specifying and designing systems that make use of this feature.

In this paper, we examine the problem of specifying a CA action-based design in a way that would allow automatic verification of whether it exhibits certain properties that are relevant to coordinated exception handling. Our aim is to understand what would be required of modeling exception propagation and handling in this design. Comprehension and documentation of exception propagation in non-concurrent software systems is by itself a complex issue and an active research area [19–24]. Concurrency is a serious complicating factor for exception propagation. In CA action-based design, a participant can not only raise and handle exceptions, but also spawn new actions that are, themselves, exception handling contexts involving multiple participants. What further aggravates matters is that it is possible for two or more exceptions to be concurrently raised inside an action. A model of actions and their participants must contemplate every possible combination of exceptions or, at least, explicitly point out combinations that cannot happen in practice. Moreover, it should make it possible to specify how participants react when faced with different sets of concurrently raised exceptions. Finally, since exception handling is closely related to action structuring, it should also model the nesting and composition [14] of CA actions and how these affect exception propagation and handling.

1.2 Proposed Approach

In this paper, we present an approach to modeling CA action-based design that makes it possible to automatically verify these models using a constraint solver. The main component of the proposed approach is a formal model of CA actions that specifies the structuring of a system in terms of actions, as well as information relevant to exception flow amongst these actions. This model can be directly specified using well-known specification languages, such as Alloy [25] or B [26], and automatically verified using tool sets associated with them. The proposed approach makes it possible to check whether a CA-action based software system satisfies several key properties.

This paper is organized as follows. The next section provides some background on CA actions, the B method and notation, and the Alloy specification language. Section 3 presents the proposed approach, including a description of the generic CA action model and some of the properties that it helps to verify. Section 4 formalizes the basic properties of the generic CA action model. We then illustrate the feasibility and usefulness of the proposed approach in two case studies. Section 6 reviews related work, and the last one sums up the paper and outlines directions for future work.

2 Background

In order to present our approach, we need to introduce several topics first. We begin with CA actions, a scheme for building fault-tolerant concurrent systems that employ exception handling. We then proceed to describe two formal specification languages, Alloy [25] and B [26]. These languages are examples of formal notations that can be used in combination with the approach proposed here in order to specify and verify some properties of fault-tolerant distributed systems based on CA actions. Both are similar to Z [27], declarative in nature, and supported by automated verification tools. It is important to stress, however, that they were designed with very different goals in mind.

2.1 Coordinated Atomic Actions

CA actions are a unified scheme for coordinating complex concurrent activities and supporting error recovery among multiple interacting components. It helps to decrease the overall system complexity and simplify development by structuring the system in terms of nested recovery units. A CA action is designed as a set of roles cooperating inside it and a set of resources accessed by these roles. An action starts when its roles are taken by participants. A participant abstracts away the underlying unit of concurrency, i.e., it can be a process, a thread, an active object, or any similar mechanism. In the course of the action, participants can access external resources. The latter must be accessed according to the ACID (atomicity, consistency, isolation, durability) properties and must provide means for these properties to be enforced. Action participants either reach the end of the action and produce a normal outcome or get involved in coordinated handling if one or more exceptions are raised within the action. If handling is successful the action completes by producing a normal outcome.

The CA action scheme enforces a clear difference between *internal exceptions* (which are raised in the action and have handlers inside the action) and *external exceptions*, which are signaled outside the action when the action cannot deliver the expected results. The latter are used to report partial action outcomes, abort effect, failure to achieve a consistent result by action participants, etc. Internal exceptions are encapsulated in the action, whereas external ones are visible in the action interface as they have to be dealt with by the containing action. When several exceptions are concurrently raised in a CA action, an exception resolution mechanism is used to define a resolved exception that represents all the exceptions that were raised. The resolved exception is then handled cooperatively by all the action roles. Exception resolution uses a data structure called the exception resolution graph, which maps sets of exceptions

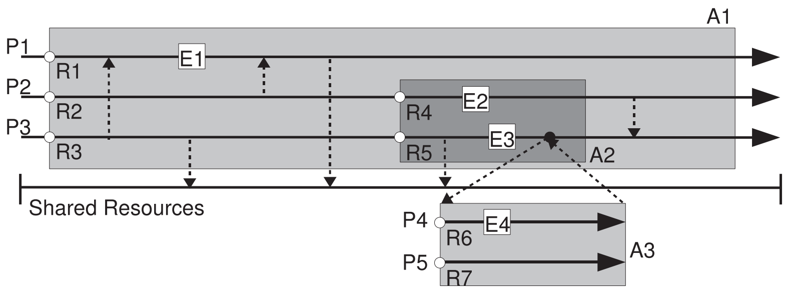

Fig. 1. A CA action example.

that can be concurrently raised to resolved exceptions. When an action cannot handle a resolved exception, its roles attempts to perform backward error recovery. If they succeed, the action is described as having *aborted*, because the system is still left in a consistent state. If the roles are unable to perform backward error recovery, the action *fails* and no guarantees can be given concerning the state of the system. In both cases, an exception is signaled to the enclosing context in order to indicate that the action did not perform as expected.

Figure 1 represents a simple system structured using CA actions. This diagram shows how the units of computation in a system interact and what information they exchange over a period of time. The top-level CA action A1 has three roles performed by participants P1, P2, and P3. Participants P2 and P3 also perform roles R4 and R5 respectively in the *nested CA action* A2. A nested action defines the exception handling context within an enclosing action and serves as a finer-grained damage confinement region. Role R5 of A2 spawns *composed CA action* A3 at some point in time before the completion of A2. R5 is interrupted from the moment A3 starts until it completes. Composed actions are started by roles in order to perform specific tasks and their lifecycles are bound to those of their spawning roles. The internal exceptions of an action are represented by small squares (labeled E1, E2, E3, and E4 in the figure). Each exception is placed near the role that it is raised by.

The fault tolerance approach to use in developing dependable systems largely depends on the fault assumptions made and on the system characteristics and requirements. In spite of all of its advantages, backward error recovery has a limited applicability. Modern systems are increasingly relying on forward error recovery and appropriate exception handling techniques [2,28]. Examples of such applications include complex systems involving human beings, COTS components, external devices, several organizations, movement of goods, operations on the environment, as well as real-time systems that do not have time to go back. Service-oriented architectures also clearly fall into this category [28]. The CA actions serve as a valuable conceptual tool in developing these systems.

2.2 The B Method

B is at the same time a formal development method and a modeling notation [26]. A formal B specification is a mathematical model of the required behavior of a system or its part, represented by a collection of modules called abstract machines. An abstract machine encapsulates a local state (local variables) and provides operations (events). The occurrence of events represents the observable behaviour of the system. The event guard defines the conditions under which the body can be executed. B statements used to describe the body of events are a mixture of executable statements (e.g. assignments or conditional statements) and abstract statements that use mathematical operations over sets and functions. The following sections offer examples and explanations of the B notation where appropriate.

The B Method supports top-down system development. In the development process, an abstract specification is transformed into an implementation following a number of correctness-preserving steps called refinements. The B method generates a number of correctness conditions, called proof obligations, for each refinement step. To guarantee that the refined system satisfies (preserves) all the specified properties, these need to be proved true, thus validating correctness. B models can also be subject to automated analysis (model checking) through the use of the ProB constraint solver [29]. However, since ProB uses undecidable logics, automated model verification must have a bounded scope in order to guarantee that verification stops.

In this work, we have chosen to use B as a specification language in order to realize the system model that we propose in Sections 3 and 4. This model comprises three components: elements of CA action-based software systems, relations and functions that connect and add information to these elements, and predicates that define rules to which valid systems must adhere.

Some features of the B notation have prompted this choice. First, it supports the definition of both types of element in a system and their instances. Moreover, both structural (actions, roles, participants) and data (exceptions) elements can be modeled as typed entities. Second, it is expressive enough to specify the formal model that we present in Section 4, including the more convoluted predicates involving transitive closures and high-order relations. Third, there is a large number of tools that support software development based on B, both entirely automated and interactive. Fourth, it supports the approach to structuring specifications that we employ. This is similar to software development based on object-oriented frameworks: part of a system consists of reusable code and design (in our case, reusable specification elements and predicates), whereas the rest of it (specification) is application-specific. The two parts are connected by the extension mechanisms provided by the underlying programming (specification) language.

2.3 Alloy Specification Language

Alloy [25] is a lightweight specification language for software design. It is amenable to a fully automatic analysis using the Alloy Analyzer (AA) [30], as well as providing a visualizer for making sense of solutions and counterexamples it finds. Alloy is based on first-order relational logic and, similarly to other specification languages such as Z [27], supports complex data structures and declarative models. Alloy aims to be a language for prototyping and verification, built with automated verification in mind from the start. In fact, Alloy was devised as the simplest formal language that could still support the creation of useful models. In this sense, it differs from Z, a more expressive language, which is intended to support (interactive) theorem proving.

In Alloy, models are analyzed within a given scope, or size (the maximum number of instances of a type). The analysis performed by the AA is sound, since it never returns false positives. Yet similarly to B, verification of Alloy models is incomplete because the AA checks are only conducted up to a certain scope. It is, however, complete up to the selected scope, i.e., the AA never misses a counterexample that is smaller than the specified scope. As pointed out in the Alloy tutorial [31], small scope checks are still useful for identifying errors. In the next sections, examples and explanations concerning the Alloy notation are given where appropriate.

The use of Alloy for specifying and verifying software systems offers many advantages: (i) it is a very simple language whose semantics is based on first-order relational logic; (ii) developers used to object-oriented languages find its syntax easy to learn; (iii) it has a very fast constraint solver; and (v) at least for small systems, the graphical counterexamples produced by the AA are easy to understand, especially when compared to execution traces. Its main shortcoming is insufficient expressiveness. One consequence of its design goals is that the language lacks several useful constructs for specifying systems, such as the notion of function or high-order relations. We understand that these constraints were intended to simplify automated verification, yet they effectively limit the ability of developers to specify real systems. In Section 5.2, we describe an example situation where this limited expressiveness makes it difficult to apply the proposed approach.

3 Proposed Approach

In order to construct robust fault-tolerant systems, it is imperative that developers start taking fault tolerance-related issues into account at the early phases of development. Our ultimate goal is to devise a general approach to rigorous development of dependable distributed systems that use both cooperative and competitive concurrency. This work specifically addresses the issue of verifying properties relevant to system structuring and coordinated exception handling in CA action-based design. The rest of this section overviews the proposed approach and briefly describes some of the properties that it helps to verify.

3.1 Overview

Figure 2 is a schematic representation of the proposed approach to verifying CA action-based software systems. Developers start by performing the traditional activities of the software development process, namely, system analysis and design. In doing that, they assume that the system is concurrent and cooperative. At the same time, they define scenarios where the system may fail (fault model), what exceptions correspond to each type of error, and where and how the exceptions are handled (exceptional activity). The system fault model and exceptional activity can be specified in the way prescribed by some studies [8]. The result of these activities is a model of the CA action-based system. This model identifies the exceptions that can be raised in each CA action and how they are handled. It is written in a language for modeling CA actions, for example, informal diagrams (as presented in Figure 1), the Coala [17] formal language, or the FTT-UML [32] profile for the UML.

To verify a CA action-based design, it is necessary to translate it into a specification language with operational semantics that allows formal verification of properties. Moreover, in accordance with the proposed approach, the language should allow both data and structural elements to be defined as typed entities that can be subtyped. In this paper we will refer to languages that meet these criteria as *specification languages*. If the language for modeling CA actions has a well-defined semantics like Coala, this translation can be completely automated by a tool. The translation can also be automated for informal notations, such as UML profiles, but only partially (syntactically). Usually some manual intervention is required to resolve ambiguities. Those developers who are more familiar with formal methods can write system descriptions directly in the specification language, and we adopt this approach in the rest of this paper. The choice of using one or two languages (one for modeling and one for verification) is based solely on usability issues.

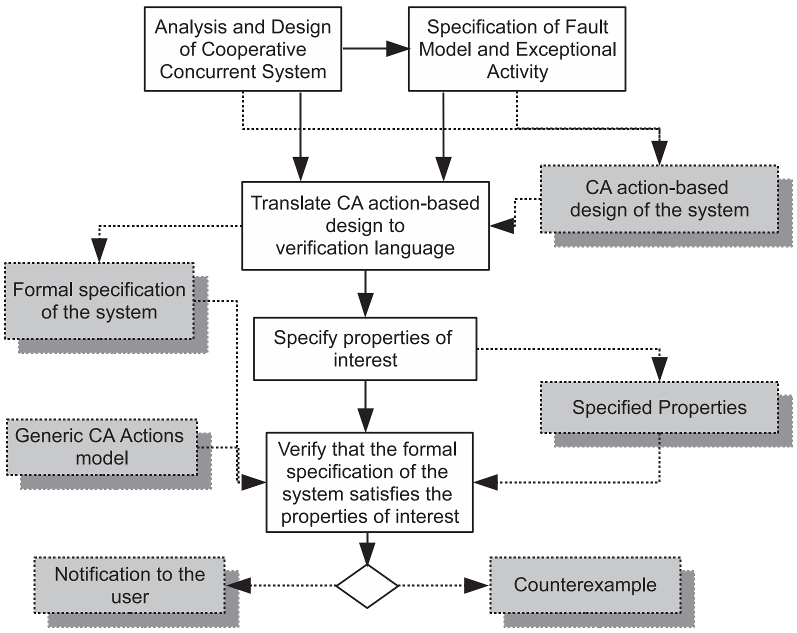
Analysis and Design of Cooperative Concurrent System
Specification of Fault Model and Exceptional Activity
Translate CA action-based design to verification language
CA action-based design of the system
Formal specification of the system
Specify properties of interest
Specified Properties
Generic CA Actions model
Verify that the formal specification of the system satisfies the properties of interest
Notification to the user
Counterexample

Fig. 2. Overview of the proposed approach. White rectangles represent activities and shaded ones stand for artifacts.

The formal specification that is produced by translating a CA action-based design into the specification language must adhere to a generic CA action meta-model, which defines the main concepts of CA actions and how they relate (hereafter referred to as *generic CA action model*). In short, this model describes an exception-handling mechanism based on CA actions, focusing on how exceptions flow amongst system components. In the overview of the generic CA action model offered in the next section, both the formal specification and the generic CA action model are described in the specification language. Up to now, we have specified generic CA action models using B and Alloy as specification languages [33]. Developers can use either of them to formalize a CA action-based design (but not both simultaneously). The purpose of having two different specification languages is to show that the proposed approach is language-agnostic. A developer who intends to employ our approach would therefore need to choose only one of them or define a generic CA action model for yet another formal notation.

To verify a system, its formal specification, together with the properties to be verified, is put into a constraint solver for the specification language. These properties are predicates that must be true for the system to be deemed well-defined or well-designed. In order to be applicable to any system description adhering to it, they are specified in terms of the elements of the generic CA action model. We have used the AA and ProB constraint solvers to verify formal specifications in Alloy and B, respectively. If any of the relevant properties does not hold, the constraint solver produces a counterexample. Besides generating a counterexample, each constraint solver includes a graphic visualizer that also helps to identify the problem.

In the rest of the paper, we focus on the three remaining activities in Figure 2, namely, “Translate CA action-based design into specification language” (assuming that the system model is written directly in the specification language), “Specify properties of interest” and “Verify that the formal specification satisfies the properties”. It is these activities that are directly related to system verification.

3.2 Generic CA Action Model

The generic CA action model formally defines an exception handling mechanism for CA action-based systems. This model can be instantiated by systems adhering to it so that certain properties related to the structure and flow of exceptions in the system can be automatically verified. It is generic in the sense that it does not depend on specific tools, formalisms or approaches to verification. As mentioned in the previous section, up to now we have specified generic CA action models using B and Alloy as specification languages. For verification we employed the constraint solvers available for these languages. An alternative to using the existing general-purpose constraint solvers would be to build a specific verification tool based on the model. The trade-off in this case is between performance (of verification) and flexibility (ability to specify new properties that need to be verified).

In our view, the structure of a system is a hierarchy of actions that contain nested actions and roles. Roles are performed by participants, units of computation such as threads and processes; they can compose additional actions that only make sense in the context of the spawning role. Hence, the main elements of the generic CA action model are actions, roles, participants, and exceptions. These elements are represented by objects of a certain type. The proposed model employs a notion of type that is compatible with that used in OO languages such as Java and C#. A type T is a set of instances, while its subtypes $T_1, T_2, ..., T_N$ of T are disjoint subsets of T. Only single inheritance is allowed.

Table 1 lists the elements of the proposed generic CA action model, i.e., the main concepts used in the definition of CA actions. An exception is any instance of type *RootException*, or some of its subtypes. The same applies to actions, roles and participants, and types *Action*, *Role* and *Participant*, respectively. We assume that instances of these types are uniquely identified by their names. The sets in the table can also be seen as unary relations and are, therefore, subject to operations that apply to relations, such as composition.

We use objects to represent exceptions rather than symbols or global variables mainly because objects are more flexible and can encode arbitrary informa-

Table 1
Basic elements of the proposed model

Element	**Description**
Action	Type that defines actions
Role	Type that defines roles of actions
Participant	Type that defines participants
RootException	Type that defines exceptions

tion on the cause of an exception [34]. Also, there are many large and complex software systems developed nowadays using object-oriented languages, such as Java and C++, which present exceptions as objects. Additionally, we avoided choosing a more usual name for the supertype of all exceptions, such as *Exception* or *Error*, in order to afford developers the flexibility to organize exceptions as required, e.g., by basing their organization on the adopted programming language. For example, considering the exception handling mechanism of Java, a developer should define at least four exception types: (i) *Throwable*, a subtype of *RootException*; (ii) *Exception*, a subtype of *Throwable*; (iii) *Error*, a subtype of *Throwable*; and (iv) *RuntimeException*, a subtype of *Exception*. An application-specific exception type would then be a subtype of one of these.

Additional information is associated with the elements of Table 1 through relations (sometimes functions). For example, a set of action roles is defined by the $Roles \in Action \times Role$ relation, which associates actions with their respective roles. The proposed model defines 16 different relations that specify three different aspects of a CA action-based software system: (i) system structure; (ii) exception flow; and (iii) exception resolution. The well-formedness of a system adhering to the model is determined by a set of predicates, or basic *properties*, defined in terms of these relations and the elements of the model. Section 3.3 offers some examples of basic properties, while Section 4 presents a formalization of the generic CA action model.

The relationship between the generic CA action model and a system description adhering to it is similar to that between an object-oriented framework and a system that instantiates it. The model defines specific points where it can be extended: extension points include types that correspond to the elements of CA action-based systems (basic types). System descriptions instantiate the model by using element types that extend (in the sense of object-oriented inheritance) the basic types. Since the properties of interest are specified in terms of the elements of the generic CA actions model, the aforementioned relations and predicates also apply to system descriptions adhering to the generic CA action model. This is similar to a method in an OO language that has a parameter of a type T but also accepts parameters of a type T', subtype of T. This approach separates the tasks of specifying a generic CA actions model (performed only once for each specification language) from the task of specifying a system and promotes reuse of properties of interest and system specifications. Nevertheless, a developer still needs to understand the generic CA model in order to specify a system. Section 5 provides two examples of instantiation of the generic CA action model.

3.3 Properties to be verified

Properties that a system must satisfy fall into three categories: basic, desired and application-specific. Basic properties define the well-formedness rules of the model, i.e., the characteristics of valid CA actions. They specify the coordinated exception handling mechanism and how actions are organized. Below are some examples of basic properties, stated informally.

BP$_A$**.** *If a participant performs a role in a nested action, it must also perform a role in the containing action.* Participants are units of computation (threads, processes) that perform roles in CA actions. In theory, any participant can perform a role in a top-level CA action. However, for a nested CA action, the definition of CA actions requires that only participants that perform roles in the containing CA action do so in the nested one.

BP$_B$**.** *There are no cycles in action nesting.* This property states that the organization of actions in the system is hierarchical, and the graph formed by their definitions (including those of nested and composed actions) has no cycles.

BP$_C$**.** *The exception resolution mechanism of an action resolves all possible combinations of concurrent internal exceptions, unless explicitly stated otherwise.* This property guarantees that every possible combination of concurrently raised exceptions is contemplated by the exception resolution graph of each CA action. Some of these combinations must be resolvable by the mechanism. The resolution graph must also explicitly account for those combinations of concurrently raised exceptions that can happen in theory, but not in practice.

Desired properties are general properties that are usually considered beneficial, although they are not part of the basic mechanism of CA actions. They describe important requirements that most fault-tolerant software systems should meet. In general, desired properties are based on the assumption that the basic properties hold. Below are some examples:

DP$_A$**.** *Top-level CA actions have no external exceptions.* This property states that the system is, in fact, fault-tolerant. It specifies that all the exceptions that reach top-level (non-nested, non-composed) CA actions are handled by these actions, and the latter do not signal any exceptions. This guarantees that the system never fails catastrophically due to unhandled exceptions. The Ariane-5 control system [35] is a classic example of a system that failed

```
predicate parts_ok() {
   (all A:Action| (all NA:A.NestedActions|
    all NAR:NA.Roles |
     !(all P:Participant|!(NAR in P.RolesPlayed
       && some (P.RolesPlayed & A.Roles)))))
}
```

Fig. 3. Alloy specification of property BP_A.

to meet this desirable property (it failed due to an uncaught exception), resulting in a very expensive accident.

DP_B. *All internal exceptions of an action are handled within it, and therefore no exceptions are propagated.* This is a desirable property that is hard to meet in practice. It states that the system adheres to the principle that an error should be handled as close as possible to its detection site [1]. For CA actions, it means that every action in the system is capable of handling all the internal exceptions that can result from exception resolution within the action, effectively masking their occurrence from the enclosing CA action.

Application-specific properties are rules about the flow of exceptions in a specific CA action-based application. An example of an application-specific property is given in Section 5.2. The generic CA action models we have specified so far include specifications of several basic and desired properties that can be used "as-is". Developers only specify additional desired properties and application-specific properties, if any.

The rest of this section offers examples of properties of interest written in Alloy. As mentioned in Section 3.1, in order to verify a CA action-based design, it is necessary to specify relevant properties in the specification language. While they do not inherently depend on any specific language, a specification language is necessary to use generic constraint solvers, such as ProB and the AA, for verification. Therefore, the examples we present in the rest of the section could also have been written in B or, as we show in the next section, in a language-agnostic manner, without significant difference. Figure 3 presents a formal specification of property BP_A in Alloy.

Figure 3 defines an Alloy predicate named `parts_ok`. Alloy predicates are logic sentences that must be checked by the AA. In the body of the predicate, `Roles`, `NestedActions` and `RolesPlayed` are names of some relations that associate information with the elements of the system. The "." operator represents relational composition (or join). More formally, given two relations $A \subseteq T_1 \times T_2 \times ... \times T_n$ and $B \subseteq T_n \times T_{n+1} \times ... \times T_{n+m}$, $A.B$ yields a relation $C \subseteq T_1 \times T_2 \times ... \times T_{n-1} \times T_{n+1} \times ... \times T_{n+m}$. Relation C comprises all the tuples formed by combining tuples from A and B whenever the last element of a tuple from A is the same as the first of a tuple from B. For example, given $A = \{(e_1, e_2), (e_2, e_3)\}$ and $B = \{(e_2, e_4), (e_2, e_5), (e_3, e_6), (e_7, e_8)\}$, $A.B$ yields $C = \{(e_1, e_4), (e_1, e_5), (e_2, e_6)\}$. In the Alloy predicate presented in Figure 3,

```
all A1:Action | ((all A2:Action |
 !(A1 in A2.NestedActions)) && (all R:Role |
  !(A1 in R.ComposedActions))) =>
   (no A1.External)
```

Fig. 4. Alloy specification of property DP_A.

`A.NestedActions` yields the set of actions nested within action `A`, assuming that `A` $\in$ `Action`, where `Action` is a type, and `NestedActions` is a relation associating actions with their nested actions. Predicate `parts_ok` states that every role of every nested action is performed by a participant that also performs a certain role in the enclosing action. The operators `all`, `!`, `&&`, and `&` represent universal quantifier, logical negation, logical conjunction and set intersection, respectively. The `some` keyword yields true if its argument is a non-empty set.

The snippet in Figure 4 shows a formal specification in Alloy for property DP_A. It states that an action that is not nested within another action and not composed by some role (i.e., top-level CA actions) has no external exceptions. Operator `=>` represents logical implication.

4 Formalization of the Generic CA Action Model

In this section, we formally specify the basic properties of the generic CA action model, using a combination of the basic set theory and relational logic. This formalization is compatible with both B and Alloy. In the latter case, due to the inability of Alloy to specify high-order quantifications (Section 5.2), it requires some minor adjustments. Our account covers three aspects of CA action-based software systems: (i) system structure, (ii) exception flow and (iii) exception resolution. These are explained in Sections 4.1, 4.2, and 4.3, respectively.

4.1 System Structure

As mentioned in Section 3.2, we consider the structure of a system to be a hierarchy of actions that contain nested actions and roles. System structure is specified in terms of four relations:

- $Roles \in Action \leftrightarrow Role$
- $NestedActions \in Action \leftrightarrow Action$
- $RolesPlayed \in Participant \leftrightarrow Role$
- $ComposedActions \in Role \leftrightarrow Action$

Table 2
Properties that define valid structuring of a CA action-based system.

Property	**Constraint**
$BP1$	$\forall A \in \mathit{Action} \bullet \lvert\{A\}.\mathit{Roles}\rvert > 0$
$BP2$	$\mathrm{ran}(\mathit{Roles}) = \mathrm{ran}(\mathit{RolesPlayed})$
$BP3$	$\forall A \in \mathit{Action} \bullet \forall P \in \mathit{Participant} \bullet \lvert\{P\}.\mathit{RolesPlayed} \bigcap \{A\}.\mathit{Roles}\rvert \leq 1$
$BP4$	$\forall R \in \mathit{Role} \bullet \lvert\mathit{Roles}.\{R\}\rvert = 1$
$BP5$	$\forall A \in \mathit{Action} \bullet \forall NA \in \{A\}.\mathit{NestedActions} \bullet \forall NAR \in \{NA\}.\mathit{Roles} \bullet$ $\exists P \in \mathit{Participant} \bullet NAR \in \{P\}.\mathit{RolesPlayed} \wedge$ $\{P\}.\mathit{RolesPlayed} \bigcap \{A\}.\mathit{Roles} \neq \{\}$
$BP6$	$\forall CA \in (\mathit{Action}.\mathit{Roles}).\mathit{ComposedActions} \bullet \forall P \in \mathit{Participant} \bullet$ $\{P\}.\mathit{RolesPlayed} \bigcap \{CA\}.\mathit{Roles} \neq \{\} \Rightarrow \neg(\exists A \in \mathit{Action} \bullet A \neq CA \wedge$ $\{A\}.\mathit{Roles} \bigcap \{P\}.\mathit{RolesPlayed} \neq \{\} \wedge A \notin \{CA\}.*\mathit{NestedActions})$
$BP7$	$\mathrm{ran}(\mathit{NestedActions}) \bigcap \mathrm{ran}(\mathit{ComposedActions}) = \{\}$
$BP8$	$\forall A \in \mathit{Action} \bullet A \notin \{A\}.*(\tilde{}(\mathit{Roles}.\mathit{ComposedActions} \bigcup \mathit{NestedActions}))$

Given an action A (an instance of type *Action*), expressions $\{A\}.\mathit{Roles}$ and $\{A\}.\mathit{NestedActions}$ yield the set of roles of action A and the set of actions nested within A, respectively. Similarly, given participant P and role R, $\{P\}.\mathit{RolesPlayed}$ and $\{R\}.\mathit{ComposedActions}$ yield the set of roles that P performs and the set of actions that R composes, if any, respectively. Table 2 lists some constraints on relations *Roles*, *NestedActions*, *RolesPlayed* and *ComposedActions*. These constraints specify properties that need to be exhibited by a system specification adhering to the generic CA action model. Each one is identified by a name matching pattern BPX, where "BP" stands for basic property and "X" is a positive integer. Properties $BP1$, $BP2$, $BP3$, and $BP4$ specify the following fundamental constraints: (1) every action has at least one role; (2) every role of every action is performed by some participant; (3) a participant plays at most one role in any given action; (4) each role is part of exactly one action. In the table, the "ran" operator yields the range of a relation. Property $BP5$ specifies that all the roles in a nested action are performed by participants who also play a role in the enclosing action. It is a language-agnostic formalization of property BP_A, specified both both informally and formally (in Alloy) in Section 3.3. $BP6$ specifies a similar constraint that targets specifically composed actions. It states that participants that perform roles in a composed action only perform roles in other actions if the latter are nested within the composed one.

$BP7$ is a simple property specifying that nested actions cannot be composed and vice versa. However, it does not preclude composed actions from having nested actions. Conversely, it does not restrict the roles of nested actions from spawning composed actions. Property $BP8$ specifies that a valid system has no action nesting or composition cycle. It is a formal definition of property BP_B (Section 3.3). It considers a CA action-based design to be a graph where actions are vertices and there is an edge between two arbitrary vertices A and B if (i) $B \in \{A\}.\mathit{NestedActions}$ or (ii) $B \in (\{A\}.\mathit{Roles}).\mathit{ComposedActions}$. If there is a cycle in this graph, the system is considered invalid. In the table, the "*" and "˜" operators stand for transitive closure and the inverse relation, respectively.

4.2 Exception Flow

Exception flow is specified in terms of twelve different relations. Six of them indicate how exceptions flow amongst actions:

- $Internal \in Action \leftrightarrow RootException$
- $External \in Action \leftrightarrow RootException$
- $AbortException \in Action \rightarrow RootException$
- $FailException \in Action \rightarrow RootException$
- $Resolution \in Action \rightarrow (POW(RootException) \rightarrow RootException)$
- $Excluding \in Action \leftrightarrow POW(RootException)$

The remaining six specify how exception flow works for roles of actions:

- $Raises \in Role \leftrightarrow RootException$
- $Generates \in Role \leftrightarrow RootException$
- $Signals \in Role \leftrightarrow RootException$
- $Masks \in Role \leftrightarrow RootException$
- $Aborts \in Role \leftrightarrow RootException$
- $Propagates \in Role \rightarrow (RootException \rightarrow RootException)$

Relations $AbortException$, $FailException$, $Resolution$, and $Propagates$ are in fact partial functions. In the rest of this subsection, each of these relations is explained in more detail. We begin by describing the relations associated with roles and then explain those that refer to actions.

Relations and properties connected with roles

We use two distinct relations to indicate the exceptions that each role is capable of handling. We are only interested in the effect the handler has on the flow of exceptions, whether it stops exception propagation or not; modeling the behavior of the actual exception handlers is beyond the scope of this work. The $Masks$ relation specifies exceptions that are masked by a role. By "masked" we mean that the component can take an action that stops the propagation of the exception and makes it possible for the system to resume its normal activity. In the scope of cooperative concurrent systems based on CA actions, an action A is capable of effectively masking an exception E if and only if, for every one of its roles $R \in \{A\}.Roles$, $E \in \{R\}.Masks$. The $Propagates$ relation describes exception handlers that do not stop the propagation of exceptions. These handlers end their execution by signaling the same exception or a new

Table 3
Properties that describe a valid flow of exceptions amongst the elements of a system.

Property	**Constraint**
$BP9$	$Masks \bigcap Aborts = \{\}$
$BP10$	$Aborts \bigcap \{R, E \mid R \in \text{dom}(Propagates) \land E \in \text{dom}(\text{union}(\{R\}.Propagates))\} = \{\}$
$BP11$	$Masks \bigcap \{R, E \mid R \in \text{dom}(Propagates) \land E \in \text{dom}(\text{union}(\{R\}.Propagates))\} = \{\}$
$BP12$	$Raises = Generates \bigcup ComposedActions.External$
$BP13$	$\forall A \in Action \bullet \forall R \in \{A\}.Roles \bullet \{R\}.Signals = (\text{ran}(\text{union}(\{A\}.Resolution)) \setminus \{R\}.Masks \setminus \text{dom}(\text{union}(\{R\}.Propagates))) \bigcup (\text{ran}(\text{union}(\{A\}.Resolution))).(\text{union}(\{R\}.Propagates))$
$BP14$	$Internal = Roles.Raises \bigcup NestedActions.External$
$BP15$	$External = Roles.Signals \bigcup AbortException \bigcup FailException$
$BP16$	$\forall A \in Action \bullet (\exists E \in RootException \bullet E \in \text{ran}(\text{union}(\{A\}.Resolution)) \land (\forall R \in \{A\}.Roles \bullet E \in \{R\}.Aborts)) \Leftrightarrow \{A\}.AbortException \neq \{\}$
$BP17$	$\forall A \in Action \bullet (\exists E \in \text{ran}(\text{union}(\{A\}.Resolution)) \bullet \neg(\exists E' \in RootException \bullet \{E\}.\text{union}((\{A\}.Roles).Propagates) = \{E'\} \land ((\forall R \in \{A\}.Roles \bullet E \in \text{dom}(\text{union}(\{R\}.Propagates))) \lor E = E')) \land \neg(\lvert(\{A\}.Roles \lhd \{R\}.Masks) \rhd \{E\}\rvert + 2 \leq \lvert\{A\}.Roles\rvert) \land E \notin \{A\}.AbortException) \Rightarrow \lvert\{A\}.FailException\rvert = 1$

one. $Propagates$ specifies a cause-consequence relationship between an exception that a role catches and one that it signals. Given a role R and exceptions E and E', if $\{E\}.\text{union}(\{R\}.Propagates) = E'$, we say that role R propagates exception E' from exception E and handles E by propagating it. The "union()" operator that appears above represents the generalized union over a set of sets. We employ it to obtain a set of instances from a set of sets of instances of the same type. For example, if $Propagates = \{R1 \mapsto \{E1 \mapsto E2\}, R2 \mapsto \{E3 \mapsto E4\}\}$, the expression $\{R1, R2\}.Propagates$ yields $\{\{E1 \mapsto E2\}, \{E3 \mapsto E4\}\}$, whereas $\text{union}(\{R1, R2\}.Propagates)$ yields $\{E1 \mapsto E2, E3 \mapsto E4\}$.

When a role is not capable of appropriately handling an exception, it might still be able to fail gracefully by returning to a state that is guaranteed to be consistent, through using a backward error recovery mechanism. The *Aborts* relation indicates whether a role can perform backward error recovery upon receipt of an exception. If, for a role R and an exception E, $R \mapsto E \in Aborts$, we say that role R aborts on exception E. Properties $BP9$, $BP10$ and $BP11$ specify that the sets of exceptions that roles mask, propagate from and abort on are disjunct. Properties $BP10$ and $BP11$ are more complex than $BP9$ because $Propagates$ is a $Role \rightarrow (RootException \rightarrow RootException)$ function, whereas $Masks$ and $Aborts$ are $Role \leftrightarrow RootException$ relations. The "dom" operator yields the domain of a relation.

The generic CA action model uses three different relations to describe the throwing of exceptions. The *Raises* relation lists the exceptions that each role can raise within its parent action. The action treats these exceptions as internal exceptions. Property $BP12$ defines this relation as the conjunction of the exceptions generated by each role and the external exceptions of the actions that it composes. The *Generates* relation specifies the exceptions generated by roles when erroneous conditions are detected. These conditions are dependent on the semantics of the application and on the assumed fault model. For reasoning about exception flow, it is the fact that the exception was raised that is important rather than the error that caused an exception to be raised. Finally, the *Signals* relation associates roles with the exceptions they throw when unable to mask a resolved exception. The exceptions signaled by a role are considered external exceptions of the parent action. Property $BP13$ of Table 3 defines *Signals* in terms of three relations: *Masks*, *Propagates* and *Resolution*. The latter is explained in the next subsection. Essentially, the set of exceptions signaled by a role comprises (i) the resolved exceptions that it does not handle (by either masking or propagating) and (ii) the exceptions it propagates.

Relations and properties associated with actions

For actions, the most important relations relevant to exception flow are *Internal* and *External*. The *Internal* relation specifies the exceptions that each action raises internally. Conversely, *External* specifies what exceptions an action signals to enclosing actions or spawning roles (in the case of composed actions). Properties $BP14$ and $BP15$ of Table 3 give definitions for the *Internal* and *External* relations, respectively. The set of internal exceptions of an action comprises the exceptions that its roles raise combined with the external exceptions of its nested actions. The set of external exceptions of an action is composed by the exceptions that its roles signal combined with the exceptions on which it fails or aborts. Relation *AbortException* specifies the exceptions on which the actions in the system are capable of aborting. An action aborts on an exception when it is unable to handle the exception, but every one of its roles is capable of returning to a consistent state through backward error recovery upon receipt of that exception. The *FailException* relation associates actions with the exceptions that they signal when they fail. An action fails when it is unable to signal an exception and does not implement a backward error recovery mechanism. Property $BP16$ of Table 3 defines constraints on the *AbortException* relation. It specifies that the roles of an action can perform backward error recovery upon receipt of a certain resolved exception if an only if the action is capable of aborting. The fact that an action A is capable of aborting is represented by associating it with an exception in the *AbortException* relation.

As pointed out in Section 2, a resolved exception is one that results from exception resolution, the process that translates a set of concurrently raised exceptions into a single exception representing multiple errors. The exception resolution graph of an action maps each set of exceptions that can be concurrently raised within the action to an exception that the roles of the corresponding action can attempt to handle. The *Resolution* relation specifies the exception resolution graph for each action in a system. It is possible that not all the combinations of exceptions that can be raised by each role can actually be raised concurrently. For example, even though roles R_1 and R_2 raise exceptions E_1 and E_2, respectively, they never do so at the same time. The *Excluding* relation explicitly states which potential combinations of internal exceptions cannot be raised concurrently at runtime. This relation is necessary because a valid action must work properly for any combination of internal exceptions that can be concurrently raised, unless explicitly stated otherwise. Exception resolution is discussed in more detail in the next section.

Property BP_{17} places constraints on the $FailException$ relation. This property specifies the sufficient conditions for the existence of an exception whose purpose is to indicate the catastrophic failure of an arbitrary action A. More specifically, it states that if the roles of A receive a resolved exception RE, there is a certain exception E such that $A \mapsto E \in FailException$ if (i) action A does not abort on RE; (ii) upon receipt of RE, two or more roles of A signal (do not mask) exceptions, and these exceptions are distinct. The two distinct exceptions may be signaled because (a) at least two different roles of A, upon receipt of E, propagate distinct exceptions, or (b) at least one role of A does not mask or propagate E, and at least one other role propagates a different exception upon receipt of E. The "$\lhd$" operator in property BP_{17} stands for domain restriction. Given a set S and a relation R, $S \lhd R$ is the set of ordered pairs $x \mapsto y$ of R whose "x" element is also an element of S. For example, given $S = \{a, b\}$ and $R = \{a \mapsto c, e \mapsto b, b \mapsto f, d \mapsto g\}$, the domain restriction $S \lhd R$ is the set of pairs $\{a \mapsto c, b \mapsto f\}$. In the same vein, the "$\rhd$" operator stands for range restriction.

4.3 Exception Resolution

As mentioned in the previous section, the *Resolution* relation is associated with exception flow. However, an exception resolution graph is a complex data structure whose well-formedness depends on conditions that are not directly related to exception flow. Therefore, we believe that it makes sense to separately describe the predicates that specify valid exception resolution graphs and their relationship with actions. We present these predicates in Table 4. Property $BP18$ specifies that every action has an exception resolution graph. Property $BP19$ imposes constraints on the domains of valid exception resolution graphs. It states that exception resolution within an action must involve only exceptions that can be actually raised by the roles of the action, or external exceptions of its nested actions. The same applies to sets of exceptions explicitly excluded from exception resolution, as specified by the *Excluding* relation. Property $BP20$ states that a set of exceptions cannot at the same time be resolved by the exception resolution graph of an action and

Table 4
Properties specific to exception resolution graphs.

Property	**Constraint**
$BP18$	$Action = \mathrm{dom}(Resolution)$
$BP19$	$\forall A \in Action \bullet \mathrm{union}(\mathrm{dom}(\mathrm{union}(\{A\}.Resolution)) \bigcup \{A\}.Excluding)$ $\subseteq (\{A\}.Roles).Raises \bigcup (\{A\}.NestedActions).External$
$BP20$	$\forall A \in Action \bullet \mathrm{dom}(\mathrm{union}(\{A\}.Resolution)) \bigcap \{A\}.Excluding = \{\}$
$BP21$	$\forall A \in Action \bullet \forall ES \in \mathrm{dom}(\mathrm{union}(\{A\}.Resolution)) \bigcup \{A\}.Excluding \bullet$ let $RE = \{RR \mid RR \in POW((\{A\}.Roles \lhd Raises) \rhd ES)\}$, $NE = \{NN \mid NN \in POW((\{A\}.NestedActions \lhd Exernal) \rhd ES)\} \bullet$ $\exists ER \in RE, EN \in NE \bullet \lvert ER \rvert + \lvert EN \rvert = \lvert ES \rvert \ \wedge \ \mathrm{ran}(ER) \bigcup \mathrm{ran}(EN) = ES$ $(\forall r \in \mathrm{dom}(ER) \bullet \lvert\{r\}.ER\rvert = 1) \ \wedge \ (\forall e \in \mathrm{ran}(ER) \bullet \lvert ER.\{e\}\rvert = 1) \ \wedge$ $(\forall e \in \mathrm{dom}(EN) \bullet \lvert EN.\{e\}\rvert = 1) \ \wedge \ (\forall e \in \mathrm{ran}(EN) \bullet \lvert EN.\{e\}\rvert = 1)$
$BP22$	$\forall A \in Action \bullet \forall ES \in POW(\{A\}.Internal) \bullet$ $ES \in \mathrm{dom}(\mathrm{union}(\{A\}.Resolution)) \ \vee \ ES \in \{A\}.Excluding$

be excluded from it.

Property $BP21$ of Table 4 is a formal specification of property BP$_C$ of Section 3.3. It states that, for any set ES of concurrently raised exceptions in the exception resolution graph of an action A, each exception in the set must have been raised by a different role or nested action of A and none of them may have contributed more than one element to ES. This property guarantees that exception resolution graphs are consistent, i.e.,, they do not depict impossible situations, such as a single role raising two or more exceptions at the same time (though it might potentially raise many different exceptions). Property $PB21$ uses a special keyword, "let", to define a macro. This notation means that, wherever the comma-separated identifiers immediately following the "let" keyword appear in the rest of the predicate, they should be replaced by the expressions following the "=" operator. For example, in property $BP21$, the expression $\{RR \mid RR \in POW((\{A\}.Roles \lhd Raises) \rhd ES)\}$ should be used wherever the identifier RE appears.

Below is an example to illustrate property $BP21$. Given an action A_1, roles R_1, R_2, R_3, and exceptions E_1, E_2, E_3, E_4, let us assume the following:

- $\{A_1\}.Roles = \{R_1, R_2, R_3\}$,
- $\{A_1\}.NestedActions = \{\}$
- $Raises = \{R_1 \mapsto E_1, R_2 \mapsto E_2, R_2 \mapsto E_3, R_3 \mapsto E_4\}$
- $Excluding = \{\}$

In this setting, if $dom(union(\{A_1\}.Resolution)) = \{\ \{E_1\}, \{E_2\}, \{E_3\}, \{E_4\}, \{E_1, E_2\}, \{E_1, E_3\}, \{E_1, E_4\}, \{E_2, E_4\}, \{E_3, E_4\}, \{E_1, E_2, E_4\}, \{E_1, E_3, E_4\}\ \}$, then A_1 will be valid according to BP_{21}, since (i) every possible combination of concurrently raised exceptions is contemplated by the exception resolution graph; and (ii) it does not include any combinations of exceptions that cannot be concurrently raised. However, if the set $\{E_1, E_2, E_3\}$ was also an element of $dom(union(\{A_1\}.Resolution))$, the resolution graph of A_1 would not be valid because exceptions E_2 and E_3 cannot be raised concurrently: they can only be raised by role R_2, which can raise only one exception at a time.

5 Case Studies

In this section we present two case studies we have conducted in order to assess the proposed approach. The first one consists in applying the proposed approach to one of the CA actions of the well-known Fault-Tolerant Production Cell system [36,37], which has been thoroughly studied in the CA action literature [38,15,39,16]. The second case study examines the usefulness of the proposed approach to formally modeling and verifying the informal CA action-based design of an embedded control system for treating patients with diabetes. The system, called the Fault-Tolerant Insulin Pump Therapy, has strict dependability requirements [40].

5.1 Fault-Tolerant Production Cell

The Fault-Tolerant Production Cell is a factory control system responsible for production of forged metal plates. The production cell comprises six electromechanical devices [37]: two conveyor belts (a feed belt and a deposit belt), an elevating rotary table, two presses, and a rotary robot that has two orthogonal extensible arms equipped with electromagnets. These devices are connected to a set of sensors that provide useful information to a controller and a set of actuators through which it can control the whole system. The task of the cell is to get a metal blank from its "environment" via the feed belt, transform it into a forged plate by using a press, and then return it to the environment via the deposit belt.

As mentioned previously, various case studies presenting CA action-based design of the Fault-Tolerant Production Cell have been published in recent years [38,15,39,16]. Therefore, it is natural that we use part of it as a case study to illustrate our own approach to developing CA action-based systems. To keep the presentation brief, we focus on a single CA action, LoadPress1, and the actions nested within it. This action was partially explained elsewhere by Xu et al. [37]. We use their partial description of the system as the basis for the application of our approach to LoadPress1.

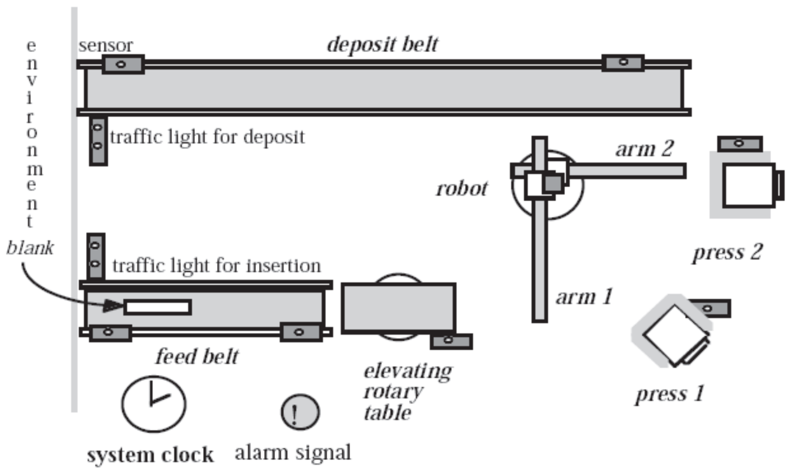

Fig. 5. Schematic view of the Fault-Tolerant Production Cell.

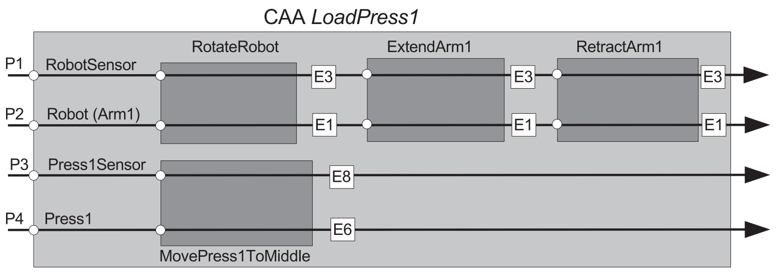

Fig. 6. Partial CA action-based design of the Fault-Tolerant Production Cell.

CA Action-Based Design

The LoadPress1 CA action controls the extensible arms of the robot in order to get a blank from the rotary table and put it on press #1 (that will be called Press1 here). Figure 6 presents a diagram of this CA action. For simplicity, it does not show accesses to shared resources or interactions between participants. The structure of the action indicates the workflow of its execution. First, the robot rotates to a position where it can get a blank from the table, uses the magnet of its first arm to get the blank, and rotates to a position where it can reach Press1 (CA action RotateRobot). At the same time, Press1 moves to its middle position so that it can receive the blank (CA action MovePress1ToMiddle). The robot then extends its first arm and drops the blank on Press1 (CA action ExtendArm1). After that, it retracts the arm and returns to its original position (CA action RetractArm1).

Various types of exception can occur within LoadPress1. These exceptions are related to either the robot or the press. As shown in Figure 5, the production cell involves two presses. For simplicity, we consider that the second press is redundant and only activated when the first one fails, in order to prevent an interruption in the processing of metal blanks. In practice, however, one would expect the two presses to work concurrently, so that a failure of one of the

Table 5
Exceptions in the CA action-based design of the Fault-Tolerant Production Cell

Exc.	**Description**
E1	Failure of the robot's position or rotary sensors
E2	Retract or extend motor does not respond
E3	Arm 1 magnet fails
E4	Robot's rotary motor fails
E5	The robot has a stuck or lost blank
Arm1Failure	Generic exception type to denote a failure of the first arm of the robot
E6	Failure of Press1's blank or position sensors
E7	Press1's motor fails
E8	Press1 has a stuck or lost blank
Press1Failure	Generic exception type to denote a failure of Press1

presses would result in a degraded service mode.

In this case study, the system is able to successfully mask a failure of one of the presses. An error in any other system component, however, means that the production cell is unable to perform. Error handlers then attempt to avoid catastrophic failure by leaving the cell in a safe state, e.g., robot arms retracted, robot turned off, presses turned off, etc. Whenever an exception propagates to a top-level CA action, an alarm is activated to notify the human operators of the error. Table 5 lists exceptions that can occur within LoadPress1. Exception E2 was originally "retract motor fails". However, since the robot arms have motors for both retracting and extending, we have made an addition to the original description. For brevity, in the model shown in Figure 6 and in the rest of this section, we assume that only four amongst the 10 exceptions in Table 5 are internal to CA action LoadPress1: E1, E3, E6 and E8. Also, we assume that at most two exceptions can be raised concurrently within LoadPress1 and that no exceptions can be raised within the CA actions nested within LoadPress1.

Applying the Proposed Approach

We modeled CA action LoadPress1 in B using the proposed approach. The snippet in Figure 7 presents part of the resulting specification. This specification is written in the Abstract Machine Notation [26], an ASCII notation for B. The `MACHINE` clause specifies the name of the B machine, i.e., of the (sub)system that this specification models. The `SETS` clause specifies the possible types of element in a model by means of B carrier sets. A carrier set in B defines a set of data elements whose internal representation is not important. In our approach, we employ carrier sets to define types of both structural elements (actions, roles, participants) and exceptions. A B carrier set is akin to the given sets in Z [27]. The `SETS` clause in the specification of Figure 7 states

```
MACHINE FTProdCell
/* System descriptions should modify the elements of
    the sets ACTION, PARTICIPANT, ROLE, and
    ROOT_EXCEPTION and the initialization. */
SETS
   ACTION = {LoadPress1,...};
   ROLE = {RobotSensor, RobotArm, ...};
   ROOT_EXCEPTION={E1, E3, Arm1Failure, GeneralFailure,...};
   PARTICIPANT = {P1, P2, ...}; ...
VARIABLES
   Internal, External, Roles, NestedActions,
   Signals, Raises,...
INVARIANT
   Roles:ACTION <-> ROLE & External:ACTION <-> ROOT_EXCEPTION
   & Resolution:ACTION +-> (POW(ROOT_EXCEPTION)
       +-> ROOT_EXCEPTION) & ...
INITIALISATION
   Roles := {LoadPress1|->RobotSensor,
       LoadPress1|->RobotArm,...} ||
   External := {LoadPress1|->Arm1Failure,
       LoadPress1|->GeneralFailure,...} ||
   Internal := {LoadPress1|->E1, LoadPress1|->E3,...} ||
   NestedActions  := {LoadPress1|->RotateRobot,...} ||
   Resolution := {LoadPress1|->{{E3, E6}|->GeneralFailure,
     {E1, E3}|->Arm1Failure,...}} ||
   Signals := {RobotArm|->GeneralFailure,
     RobotArm|->Arm1Failure,...} ||
   Raises := {RobotArm|->E3,...} ||
   Generates := {RobotArm|-> E3,...} ||
   Excluding := {LoadPress1|->{E1, E6, E8},...} || ...
OPERATIONS
   ...
END
```

Fig. 7. B specification of the Fault-Tolerant Production Cell.

that there is one action in the system called `LoadPress1`, two roles named `RobotSensor` and `RobotArm`, and so on.

The `VARIABLES` clause, which in B is employed to specify variables of a model, specifies the relations defining the proposed exception flow model. Type constraints for these variables are specified by the `INVARIANT` clause. For example, it states that `Roles` is an $Action \leftrightarrow Role$ relation. Both clauses are part of the generic CA action model. imported by different system descriptions. This subject is further discussed in the next section. The `INITIALISATION` clause assigns values to the variables defined under `VARIABLES`. In the example of Figure 7, it states that action `LoadPress1` has roles named `RobotSensor` and `RobotArm`. It also states that `LoadPress1` has at least two external exceptions: `GeneralFailure` and `Arm1Failure`. We briefly explain our use of the `OPERATIONS` clause later in this section.

Since we have applied the proposed approach to a partial (purposefully incomplete) specification [37], some of the problems we encountered may have been addressed in subsequent work on the Fault Tolerant Production Cell. Nonetheless, it is worth pointing out that the application of the proposed approach did identify several important issues that should have been included in the system specification. For example, the original specification does not clarify what should be done when two or more devices fail concurrently (e.g., Press1 fails and the robot fails as well). However, as can be seen from the structure of the system, this is a real possibility (e.g., E1 and E8 from Figure 6 could be raised concurrently within LoadPress1) and our generic CA action model requires that this is addressed. Therefore, we introduced an additional exception, naming it GeneralFailure, that is signaled by LoadPress1 to its enclosing context. As its name indicates, it signals a generalized failure in the production cell (e.g., because both the press and the robot failed).

Additionally, the application of the proposed approach highlighted the need to understand how the internal and external exceptions of LoadPress1 relate. For example, the action has an external exception Press1Failure and an internal one press1_failure. It is not, however, clear from the system specification what each one means and, most importantly, what (if any) is the causality relation between them. We claim that the proposed approach highlights this issue because it requires us to explicitly indicate the internal exceptions that, when raised, might result in the signaling of a given external exception. If this information is not given in a B specification adhering to the proposed approach, ProB will complain during verification.

We specify the basic properties of the generic CA action model in B under the `OPERATIONS` clause of the B machine. Each operation evaluates a guarded condition (corresponding to the conjunction of some of the basic properties) and assigns the value "yes" to an auxiliary variable if it is true, and "no" otherwise. Under the `INVARIANTS` clause, we specify an invariant that says that each such auxiliary variable must always have the "yes" value. Therefore, if a basic property is violated, the value "no" will be assigned to one of the auxiliary variables, and ProB will point out an invariant violation. For example, if we modify the specification presented of Figure 7 so that the role `RobotArm` raises exception `E3` but does not generate it, ProB detects an invariant violation due to the `rolesConsistent` operation (Figure 8).

The B snippet in Figure 8 defines an operation named `rolesConsistent` that does not take any input parameters. The body of the operation consists of a guarded command. The guard specifies that, for every action `Act` and every role `R` of `Act` (operator "!" indicates universal quantification), the set of

```
rolesConsistent =
   IF !Act.((Act:ACTION) =>
     !R.((R:ROLE & R:Roles[{Act}]) => ...
        & (Raises[{R}] = External[ComposedActions[{R}]] \/
          Generates[{R}])
    ) )
   THEN RolesConsistent := yes
   ELSE RolesConsistent := no
   END;
```

Fig. 8. A B operation specifying a property of interest.

```
   IF !Act.( (Act : ACTION) => ...
      & (#E.(E:ROOT_EXCEPTION & E:ran(union(Resolution[{Act}])) &
              (!R.((R:ROLE & R:Roles[{Act}])=> E:Aborts[{R}]))
           ) => (#AE.(AE:ROOT_EXCEPTION &
                   AE = AbortException(Act)))
        )
      )
   THEN ActionsConsistent := yes
   ELSE ActionsConsistent := no
   END;
```

Fig. 9. B specification of property $BP16$.

exceptions that R raises comprises the exceptions that it generates combined (\/ is the set union operator) with the set of exceptions signaled by actions that it composes. If the guard evaluates to true, the "yes" value is assigned to the RolesConsistent auxiliary variable, indicating that no invariant violations occurred. Otherwise, the variable receives the "no" value, signaling a violation. In the example of Figure 8, the notation A[{B}], where B is a single element and A is a relation, is equivalent to {B}.A, where "." represents a relational join.

The B snippet in Figure 9 presents the specification of basic property $BP16$ (Section 4.2). This predicate specifies that, if all the roles in an action Act can perform backward error recovery upon receipt of a certain resolved exception, then the action is capable of aborting. In our generic CA action model, this is represented by the existence of a pair { Act $\mapsto$ E } $\in$AbortException, where E is an exception.

The predicate in Figure 9 states that, for each action Act in a system, if there is an exception in the exception resolution graph of Act to which a set of concurrently raised exceptions is mapped, and this exception is in the Aborts set of every role of Act, then there is an exception mapped to Act in AbortException. This mapped exception is signaled by Act to indicate to an enclosing action that it has failed but was able to perform backward error recovery. Operator "#" represents existential quantification in B.

In conducting this case study, we have analyzed a number of papers, technical reports and specific scenarios. Nevertheless, we have not found any suggestion as to what the system should do when three or more exceptions are raised concurrently within an action. At first, we assumed that this was simply due to space constraints. A more in-depth study, however, has shown a fundamental limitation in the use of CA actions. In accordance with the existing definitions of CA actions [17,10], our model requires that developers either describe an exception resolution graph comprising every combination of exceptions that can be concurrently raised within each action in a system or explicitly specify cases that cannot happen in practice. This requirement of CA actions is intended to improve fault tolerance, as it produces concurrent systems that are capable of gracefully handling any possible combination of system errors. At the same time, however, it reduces the scalability of CA actions in general and exception resolution graphs in particular. The size of the resolution graph grows exponentially with the number of exceptions that can be raised concurrently within a given context. For example, if we included all the 8 internal exceptions of Table 5 (E1-8) in our model, we would end up with more than 250 different combinations. Although there are cases that cannot happen in practice (e.g., when the same role raises two different exceptions at different moments), they still have to be addressed, one by one, by the *Excluding* relation. It should be stressed that this problem is not specific to the proposed approach, but is typical of the use of exception resolution graphs in general. Our approach simply highlights this limitation because it adheres to the definition of CA actions and requires CA action-based design to contemplate every possible combination of concurrently raised exceptions.

5.2 Fault-Tolerant Insulin Pump Therapy

The Fault-Tolerant Insulin Pump Therapy [40] (FTIPT) is a control system for treating patients with diabetes with strict reliability requirements. The system is based on the Continuous Subcutaneous Insulin Injection technique [40] and involves several sensors and actuators that must function concurrently and continuously. These sensors and actuators are wearable devices worn by patients undergoing treatment. The dose of medicine administered by the system includes two types of insulin: rapid action insulin (RAI) and long action insulin (LAI). Sensors and actuators exchange information through wireless communication channels. Sensors send information about the vital signs of a patient to a server located in a hospital. The latter forwards this information to a doctor who defines the amount of insulin to inject. The server then communicates with the actuators that use pumps to administer the established dose of insulin.

Both sensors and actuators may fail. Sensors can fail by stopping to send

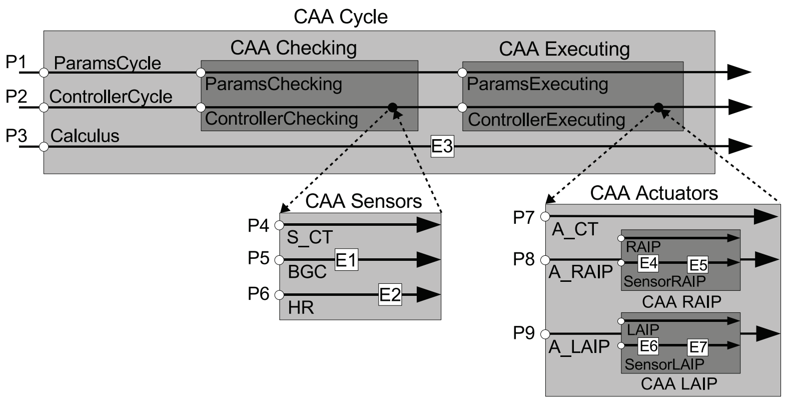

Fig. 10. CA action-based design of the Fault-Tolerant Insulin Pump Therapy

information about a patient's vital signs. However, when they do send information, the latter is assumed to be correct. Actuators can also fail - among other things, because there is not enough insulin to apply the required dose. Whenever an error is detected, treatment is interrupted and an alarm located in a remote emergency room is activated. We assume that the wireless channels do not fail.

CA Action-Based Design

Capozucca et al [40] use CA actions to design and implement the FTIPT. The system is organized as a set of actions that structure the execution of sensors and actuators. Coordinated exception handling is used as the main fault tolerance mechanism, since it is not possible to roll back when insulin has been administered to a patient. The CA action-based design devised by the authors is informal and specified using diagrams and textual descriptions.

Figure 10 presents a diagram of the system. For simplicity, it does not show accesses to shared resources or interactions between participants. CA action CAA Cycle controls the overall execution of the system and determines the amount of insulin that must be injected for each pump on the basis of the patient's vital signs. Actions CAA Sensors and CAA Actuators are spawned by roles ControllerChecking and ControllerExecuting of actions CAA Checking and CAA Executing, and are responsible for collecting the patient's vital signs and administering the insulin, respectively. Each of these composed CA actions has three roles. Roles A_RAIP and A_LAIP of CAA Actuators spawn composed CA actions CAA RAIP and CAA LAIP, respectively. The latter two control the two pumps that will administer the two types of insulin.

Seven different types of exceptions can be raised in the system (Table 6). For most of these errors, exception handling consists in stopping the treatment and activating the alarm in the emergency room. In some cases, such as when the value of a sensor cannot be obtained, the handler will try again once before

Table 6
Exceptions in the CA action-based design of the FTIPT

Exc.	Description
E1	Heart Rate (HR) sensor does not respond
E2	Blood Glucose (BGC) sensor does not respond
E3	Delivery limit reached
E4	Rapid action insulin pump (RAIP) does not respond
E5	Rapid action insulin pump (RAIP) stops during delivery
E6	Long action insulin pump (LAIP) does not respond
E7	Long action insulin pump (LAIP) stops during delivery

giving up.

Applying the Proposed Approach

We have modeled the CA action-based design described in the previous section in Alloy. The specification snippet in Figure 11 shows part of the Alloy specification of the system. Its complete specification is available elsewhere [33].

In Alloy, a signature (`sig` keyword) specifies a type. Keyword `one` indicates that a signature has exactly one instance[1]. We use signatures for modeling actions, roles, participants and exceptions (signature `Key` will be explained below). Additional information is associated with these elements using relations (Section 3.2). These relations are explicitly instantiated by facts, i.e., predicates that the AA must assume to be true when evaluating constraints. For instance, fact `SystemStructure` in the snippet in Figure 11 states, among other things, that CA action `CAAChecking` has two roles, `ControllerChecking` and `ParamsChecking`, and no nested actions. Moreover, it states that participant P1 performs roles `ControllerChecking` and `ControllerCycle` of actions `CAAChecking` and `CAACycle`, respectively. Moreover, fact `ExceptionFlow` states, among other things, that roles `BGC` and `HR` raise exceptions `E1` and `E2`, and that these are internal exceptions of CA action `CAASensors`. The `open` clause in the beginning of the specification imports the definitions of the basic types of the proposed model: `Action`, `Role`, `Participant` and `RootException`. Moreover, it imports the predicates that specify the basic properties of CA actions and certain predefined desired properties.

Fact `ExceptionResolution` in the specification in Figure 11 describes the exception resolution graph of the FTIPT. It uses subtypes `K1` and `K2` of signature `Key` to associate internal and external exceptions of action `CAASensors`. `Key`

[1] In Alloy, a type is simply a set of instances. Moreover, for the sake of uniformity and ease of use, Alloy treats instances (single elements) as unitary sets. Therefore, a singleton type can be treated as an instance of itself in a specification. For example, signature `CAACycle` defines both the homonym type and its sole instance.

```
//Imports generic CA action model
open CoordinatedExceptionHandling

//CA actions extend ‘‘Action’’, roles extend ‘‘Role’’,
//exceptions extend ‘‘RootException’’, etc.
one sig CAACycle, CAAChecking, CAASensors,
   CAAExecuting extends Action{}
one sig ControllerChecking, ParamsChecking, S_CT, BGC,
   HR extends Role {}
one sig E1, E2, E3 extends RootException {}
one sig P1, P2, P3, P4, P5 extends Participant {}
one sig K1, K2, K3 extends Key {}
...     //Other declarations.
fact SystemStructure {
   CAACycle.NestedActions = CAAChecking + CAAExecuting
   CAACycle.Roles = ControllerCycle + ParamsCycle
     + Calculus
   CAAChecking.Roles = ControllerChecking + ParamsChecking
   no CAAChecking.NestedActions
   ControllerChecking.ComposedActions = CAASensors
   CAASensors.Roles = S_CT + BGC + HR
   P1.RolesPlayed = ControllerCycle + ControllerChecking
...// Other definitions. }
fact ExceptionFlow {
   CAASensors.Internal = E1 + E2
   CAASensors.External = AlarmEXC
   BGC.Generates = E1 && BGC.Raises = E1
   HR.Raises = E2 && HR.Generates = E2
...// Other definitions. }
fact ExceptionResolution {
   CAASensors.ToResolve = E1->K1 + E2->K2
   CAASensors.Resolved = K1->AlarmEXC + K2->AlarmEXC
...// Other definitions.}
```

Fig. 11. Alloy specification of the Fault-Tolerant Insulin Pump Therapy

is an auxiliary signature defined in the Alloy version of the generic CA action model. It is necessary because the exception resolution graph of an action is a function from sets of exceptions to exceptions. Since it is not possible to define high order relations in Alloy, we used a pair of relations, with one associating internal exceptions to keys (`ToResolve`), one key for each mapping, and the other associating each key to an external exception (`ResolvedTo`). Fact `ExceptionResolution` states that exceptions `E1` and `E2` are both resolved to exception `AlarmEXC`. We would like to emphasize that this workaround used for specifying the exception resolution graph of an action is not necessary in the B version of the generic CA action model.

The positive outcome of our work on developing the formal specification of the

```
(AlarmEXC in CAACycle.Masks)
 && (all A:(Action - CAACycle)
 | AlarmEXC in A.Internal => AlarmEXC in A.External)
```

Fig. 12. An application-specific property of the FTIPT system.

FTIPT case study is that it has helped us to identify a number of shortcomings in the original informal description of the system. These were discovered in the process of formalizing and verifying the system.

According to the original system description, the handlers for exceptions `E4` and `E6` "must stop the delivery of insulin and ring the danger alarm". This statement does not identify, however, which CA action will be responsible for ringing the alarm when one of these exceptions is raised. Even though we are not explicitly modeling the actual alarm, this information is still relevant. If the alarm is to be activated by a CA action other than that where the exception was raised, an exception should be propagated from the CA action where the error was detected to the one that will ring the alarm. However, no such exception exists in the original design of the system.

For simplicity, we could assume that a certain role in the CA action where an exception is raised is responsible for ringing the alarm. However, this is not the best option since it disperses the responsibility of activating the alarm through the entire application, partially defeating the purpose of decomposing the system into actions. In the end, we decided to add a new exception named `AlarmEXC` to the system specification. This exception is signaled by actions `CAASensors`, `CAARAIP` and `CAALAIP`, and propagated all the way up to `CAACycle`, where it is handled. In our later discussion of the matter with the authors of the original case study, it transpired they had intended to make it possible to signal failures to the most external CA actions; however, the paper does not propose any means for achieving this. In order to explicitly establish that `AlarmEXC` can only be handled by `CAACycle`, we have specified this constraint as an application-specific property (Figure 12). Assuming the basic properties hold, it states that, for any action other than `CAACycle`, if `AlarmEXC` is an internal exception, it is also external. Moreover, it states that `CAACycle` handles `AlarmEXC`.

After finishing the specification of the system in Alloy, we tried to verify the basic CA action properties using the AA. In a few seconds, the latter presented a counterexample indicating that the specification failed to satisfy a relevant property. A careful analysis of the counterexample revealed that property $BP21$ of the previous section was being violated. This happened because the case where exceptions `E1` and `E2` are raised concurrently in action `CAASensors` was not covered by the exception resolution mechanism of the action. This is a direct violation of basic property $BP22$ (Section 4.3). To fix the specification, we extended the resolution mechanism of the action so that, when these two exceptions are raised concurrently, they are resolved to `AlarmEXC`. However, a discussion of this problem with the authors of the original case study revealed that these two exceptions are actually never raised concurrently in practice. Hence, we modified the Alloy specification accordingly. The generic CA action model defines a relation, `Excluding`, intended to exclude combinations of internal exceptions that are never raised concurrently from the exception resolution graph of an action. This relation is already taken into account by the basic CA properties defined by the generic CA action model. By default, no combinations are excluded. The following line was introduced in the specification of the system:

```
CAASensors.Excluding = (E1 + E2) -> K3
```

6 Related Work

Several models have been proposed for formalizing the CA action concept with the intention either to give a more complete and rigorous description of the concept or to verify systems designed using CA actions. In this section we briefly describe the most significant amongst these formalizations, comparing them to our own work. The COALA framework [17] was proposed to allow system developers to model complex distributed/concurrent systems. Within this framework a formalization of the CA action concept is developed using CO-OPN/2 [41], an object-oriented language based on Petri nets and partial order-sorted algebraic specifications. Although CO-OPN/2 specifications are amenable to mechanical verification (through translation to "regular" Petri nets), no attempt is made to specify systems properties or verify systems described in COALA. The authors' main goal is to devise a semantically precise specification language for CA actions.

Another model used for formalizing the CA action concept is the ERT (ERT stands for extraction, refusals and traces) [42]. Refusals and traces are notions that come from semantic models of CSP; the term extraction refers to a specific technique used to relate systems specified at different levels of abstraction. This model does not have as strong a focus on exception flow as ours. Hence, there are many properties that we can verify, especially those relevant to exception resolution, that cannot be directly specified using the ERT-based model.

A mathematical framework, based on Timed CSP, for representing the use of CA actions in real-time safety-critical systems is proposed in [43]. It allows the interactions between concurrently functioning pieces of equipment to be modeled - and their behavior to be reasoned about - in an abstract way. The framework models dynamic system structuring using CA actions and explicitly uses events representing synchronization between items and the control system to allow the action context to be changed dynamically. Unlike our approach, the framework was not developed for dealing with erroneously behaving action participants. However, it helps to achieve a better understanding of the CA action concept and can be used in developing general models incorporating mechanisms that support system safety.

Tartanoglu et al. [44] have devised a formalization for CA actions using the B method. More specifically, they have formalized part of the CA action middleware infrastructure. This work is complementary to ours in that it focuses on an issue that we do not tackle: the dynamic structure of CA actions, e.g., joining participants, the start and end of nested CA actions, etc. The authors' account of how they address issues such as exception flow and resolution is, however, very sketchy. Moreover, they do not discuss how to model CA action-based applications using their approach, or how to conduct automated verification.

The concept of Dependable Multiparty Interactions (DMIs) [16] has many similarities with that of CA actions, and is formally specified using Temporal Logic of Actions [45] (TLA). There were several earlier attempts to specify the CA action semantics using TLA (for example, the one reported in [46]). However, none of them has been used to mechanically verify whether a system model satisfies certain properties. In another work [47], Bertolini et al. modeled DMIs using Stochastic Automata Networks, a formalism based on Markov chains. This formalization is complementary to ours, as it focuses on the throughput of a DMI-based system, particularly when failures occur periodically.

Xu et al. [48] use a formal approach to model and verify a safety-critical system designed using CA actions. To model-check the system controlling a fault-tolerant Production Cell, the state transition system corresponding to a CA action-based design is expressed in SMV (Symbolic Model Verifier) [49], and the properties of the system to be analyzed are expressed in CTL. This work strongly emphasizes the execution order of CA actions in a system. The authors also model the exception resolution graph for the CA actions they design, but the approach they propose does not include any means for verifying, for example, whether a resolution graph is valid. Nevertheless, this work is also complementary to ours, as it models aspects of CA action-based systems that ours does not.

Capozucca et al. [50] describe a framework for implementing systems based on CA actions. Their framework, CAA-DRIP, is an evolution of the DRIP framework proposed by Zorzo and Stroud [39]. Complementing the description in CAA-DRIP, the authors also present a partial formalization of CA actions based on statecharts [51]. This formalization focuses only on the overall states in which a CA action can be, without going into the more convoluted mechanisms of CA actions, such as the structuring of the actions that compose a system or exception resolution.

In a previous work [52], some of us have described an approach to designing and verifying an architectural view [53] that centers on how exceptions flow amongst architectural components. This work places more emphasis on the effect that different architectural styles [54] have on exception propagation than on the verification of properties associated with exception flow. Moreover, it does not deal with software systems where multiple exceptions might be raised concurrently within the same exception-handling context.

A preliminary version of this paper appeared elsewhere [55]. It does not explain the generic CA actions model (either formally or informally), presents only one case study, does not show how the proposed approach can be instantiated to the B notation, or discuss related work. In another earlier study [21] some of us defined a formal exception flow model for non-distributed software architectures. Although we use a similar approach in order to define this previous model (a generic model, applications adhering to the model and mature verification tools), the generic model itself differs greatly from the one we present in this paper. The most important difference is that for cooperative concurrent systems exception-handling contexts are much more complex. In such systems, a context must involve all the processing units that cooperate, and exception handling should also be performed cooperatively. For non-cooperative systems, contexts are localized processing units (i.e., exception handling concerns only a single component), and exceptions are either handled within them or propagated to an outer context that is also localized. Another significant difference is that exception propagation is much more complex in an exception-handling mechanism that involves cooperative handling. This shows in many parts of the formal model and, particularly, in Section 4.3, where we describe how exception resolution works. This issue does not arise for systems where only one exception at a time can be raised within an exception-handling context. Finally, it is important to stress that this previous work does not include any case studies, whereas in this paper we present two extensive case studies.

7 Concluding Remarks

We have presented an approach to specifying and verifying cooperative concurrent systems that use exception handling to achieve fault tolerance. The purpose of this approach is to guarantee that the fault tolerance mechanisms used to build a reliable system are also reliable. The main contribution of this paper is to offer a formalization of CA actions that makes it possible to automatically check whether a CA action-based design satisfies certain properties related to exception handling. The usefulness of the proposed approach has been demonstrated by two case studies. Even for simple applications, the proposed approach has helped us to uncover some implicit assumptions in the original designs of the systems. The problems we have identified are directly related to the use of exception handling. It would be harder to expose issues like that using other formal models to specify CA actions because they focus on different aspects of CA action-based systems, such as temporal ordering of events [15] or dynamic CA action structuring [44].

This work does not address all of the important aspects of systems structured as CA actions. For example, our approach does not model consistent access to external resources or the dynamic structure of nested actions. Furthermore, this work does not cover issues related to synchronization amongst action participants. In the future, we intend to expand the system model used in our approach to address these issues and provide a more comprehensive framework for verifying CA action-based systems. More specifically, we intend to devise a way to separate the specifications of the CA actions middleware from CA action-based applications. Moreover, we are currently working on a rigorous software development methodology whose emphasis is on modeling the error-handling behavior of software systems. This methodology employs the proposed approach to verification.

The Fault-Tolerant Production Cell case study has shown that the exception resolution graph approach [11] to dealing with concurrently raised exceptions is not scalable. This suggests that future research on exception handling for concurrent systems should pursue more scalable solutions to dealing with multiple exceptions being raised at the same time. An alternative would be to guarantee that certain parts of an asynchronous system are executed synchronously.

Acknowledgements

Fernando Castor Filho conducted parts of this research while supported by FAPESP/Brazil, grants #06/04976-9 and #02/13996-2. He is currently partially supported by CNPq/Brazil, grants #481147/2007-1 and #550895/2007-8. Alexander Romanovsky is supported by the EPSRC/UK TrAmS platform grant and the FP7 ICT DEPLOY integrated project. Cecília Rubira is partially supported by CNPq/Brazil, grants #301446/2006-7 and #484138/2006-5. We would like to thank the anonymous referees, who provided many useful comments and suggestions and greatly helped to improve the paper.

References

[1] T. Anderson, P. A. Lee, Fault Tolerance: Principles and Practice, 2nd Edition, Springer-Verlag, 1990.

[2] F. Cristian, Exception handling, in: T. Anderson (Ed.), Dependability of Resilient Computers, Blackwell Scientific Publications, 1989, pp. 68–97.

[3] J. B. Goodenough, Exception handling: Issues and a proposed notation, Communications of the ACM 18 (12) (1975) 683–696.

[4] W. Weimer, G. Necula, Finding and preventing run-time error handling mistakes, in: Proc. of the 19th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2004, pp. 419–433.

[5] D. Reimer, H. Srinivasan, Analyzing exception usage in large java applications, in: Proc. of ECOOP'2003 Workshop on Exception Handling in Object-Oriented Systems, 2003.

[6] S. Bodoff, The J2EE Tutorial, Addison-Wesley, 2004.

[7] F. Castor Filho, N. Cacho, E. Figueiredo, R. Ferreira, A. Garcia, C. M. F. Rubira, Exceptions and aspects: The devil is in the details, in: Proc. of the 14th ACM SIGSOFT Symposium on Foundations of Software Engineering, 2006, pp. 152–162.

[8] C. M. F. Rubira, R. de Lemos, G. Ferreira, F. Castor Filho, Exception handling in the development of dependable component-based systems, Software – Practice and Experience 35 (5) (2005) 195–236.

[9] C. Bernardeschi, A. Fantechi, S. Gnesi, Model checking fault-tolerant systems, Software Testing, Verification, and Reliability 12 (2002) 251–275.

[10] J. Xu, B. Randell, A. B. Romanovsky, C. M. F. Rubira, R. J. Stroud, Z. Wu, Fault tolerance in concurrent object-oriented software through coordinated error recovery, in: Proc. of the 25th Symposium on Fault-Tolerant Computing Systems, Pasadena, USA, 1995, pp. 499–508.

[11] R. H. Campbell, B. Randell, Error recovery in asynchronous systems, IEEE Transactions on Software Engineering SE-12 (8) (1986) 811–826.

[12] J. Gray, A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann, 1993.

[13] D. M. Beder, A. B. Romanovsky, B. Randell, C. R. Snow, R. J. Stroud, An application of fault tolerance patterns and coordinated atomic actions to a problem in railway scheduling, Operating Systems Review 34 (4) (2000) 21–31.

[14] A. Romanovsky, P. Periorellis, A. F. Zorzo, Structuring integrated web applications for fault tolerance, in: Proc. of the 6th IEEE ISADS, 2003, pp. 99–106.

[15] J. Xu, B. Randell, A. B. Romanovsky, R. J. Stroud, A. F. Zorzo, E. Canver, F. W. von Henke, Rigorous development of an embedded fault-tolerant system based on coordinated atomic actions, IEEE Transactions on Computers 51 (2) (2002) 164–179.

[16] A. Zorzo, A. Romanovsky, J. Xu, B. Randell, R. Stroud, I. Welch, Using coordinated atomic actions to design safety-critical systems: A production cell case study, Software – Practice and Experience 29 (8) (1999) 677–697.

[17] J. Vachon, N. Guelfi, Coala: a design language for reliable distributed system engineering., in: Proc. of the Workshop on Software Engineering and Petri Nets, Aarhus, Denmark, 2000, pp. 135–154.

[18] P. A. Buhr, R. Mok, Advanced exception handling mechanisms, IEEE Transactions on Software Engineering 26 (9) (2000) 1–16.

[19] N. Cacho, F. Castor Filho, A. Garcia, E. Figueiredo, Ejflow: Taming exceptional control flows in aspect-oriented programming, in: Proc. of 7th ACM Conference on Aspect-Oriented Software Development, 2008.

[20] R. Coelho, A. Rashid, A. Garcia, F. C. Ferrari, N. Cacho, U. Kulesza, A. von Staa, C. J. P. de Lucena, Assessing the impact of aspects on exception flows: An exploratory study, in: Proceedings of the 22nd European Conference on Object-Oriented Programming, Paphos, Cyprus, 2008, pp. 207–234.

[21] F. Castor Filho, P. H. da S. Brito, C. M. F. Rubira, Reasoning about exception flow at the architectural level, in: Rigorous Development of Complex Fault-Tolerant Systems, Vol. 4157 of LNCS, Springer-Verlag, 2006, pp. 80–99.

[22] C. Fu, B. G. Ryder, Exception-chain analysis: Revealing exception handling architecture in java server applications, in: Proceedings of 29th ACM/IEEE International Conference on Software Engineering, Minneapolis, USA, 2007, pp. 230–239.

[23] S. Jiang, et al., An approach to analyzing exception propagation, in: Proc. of the 8th IASTED International Conference on Software Engineering and Applications, 2004.

[24] M. P. Robillard, G. C. Murphy, Static analysis to support the evolution of exception structure in object-oriented systems, ACM Transactions on Software Engineering and Methodology 12 (2) (2003) 191–221.

[25] D. Jackson, Alloy: A lightweight object modeling notation, ACM Transactions on Software Engineering and Methodology 11 (2) (2002) 256–290.

[26] J. R. Abrial, The B-Book: Assigning Programs to Meanings, Cambridge University Press, 1996.

[27] J. Woodcock, J. Davies, Using Z: Specification, Refinement, and Proof, Prentice-Hall, Inc., 1996.

[28] F. Tartanoglu, N. L. Valérie Issarny, Alexander B. Romanovsky, Coordinated forward error recovery for composite web services, in: Proc. of the 22nd IEEE SRDS'03, Florence, Italy, 2003, pp. 167–176.

[29] M. Leuschel, M. J. Butler, Prob: A model checker for b, in: Proc. of the International Symposium of Formal Methods Europe (FME'2003), LNCS 2805, Pisa, Italy, 2003, pp. 855–874.

[30] D. Jackson, I. Schechter, I. Shlyakhter, Alcoa: The alloy constraint analyzer, in: Proc. of the 22nd ACM/IEEE International Conference on Software Engineering, Limerick, Ireland, 2000, pp. 730–733.

[31] D. Jackson, Alloy home page, available at `http://sdg.lcs.mit.edu/alloy/default.htm` (2004).

[32] N. Guelfi, G. L. Cousin, B. Ries, Engineering of dependable complex business processes using uml and coordinated atomic actions., in: Proc. of International Workshop on Modeling Inter-Organizational Systems, 2004, pp. 468–482.

[33] F. Castor Filho, A. Romanovsky, C. M. F. Rubira, Verification of coordinated exception handling, Tech. Rep. CS-TR-927, School of Computing Science, University of Newcastle upon Tyne (2005).

[34] A. Garcia, C. Rubira, A. Romanovsky, J. Xu, A comparative study of exception handling mechanisms for building dependable object-oriented software, Journal of Systems and Software 59 (2) (2001) 197–222.

[35] J. L. Lions, et al., Ariane 5 - flight 501 failure, Tech. rep., European Space Agency (1996).

[36] C. Lewerentz (Ed.), Formal Development of Reactive Systems: Case Study "Production Cell", Vol. 891 of LNCS, Springer-Verlag, 1995.

[37] J. Xu, B. Randell, A. Romanovsky, R. J. Stroud, E. Canver, Developing control software for production cell ii: Failure analysis and system design using ca actions, in: Proc. of the 3rd International Workshop on Design for Validation, Louvain-La-Neuve, 1998.

[38] E. Canver, D. Schwier, A. Romanovsky, J. Xu, Formal verification of caa-based designs: The fault-tolerant production cell, Tech. Rep. 3rd Year Report, ESPRIT Long Term Research Project 20072 on Design for Validation (November 1998).

[39] A. Zorzo, R. J. Stroud, A distributed object-oriented framework for dependable multiparty interactions, in: ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, Denver, USA, 1999, pp. 435–446.

[40] A. Capozucca, N. Guelfi, P. Pelliccione, The fault-tolerant insulin pump therapy, in: Rigorous Development of Complex Fault-Tolerant Systems, LNCS 4157, Springer, 2006, pp. 59–79.

[41] D. Buchs, N. Guelfi, Formal development of actor programs using structured algebraic petri nets., in: Proc. of the 5th International Conference on Parallel Architectures and Languages Europe, LNCS 694, Springer-Verlag, Munich, Germany, 1993.

[42] M. Koutny, G. Pappalardo, The ert model of fault-tolerant computing and its application to a formalisation of coordinated atomic actions, Tech. Rep. CS-TR 636, School of Computing, University of Newcastle upon Tyne (1998).

[43] S. Veloudis, N. Nissanke, Modelling coordinated atomic actions in timed csp, in: Proc. of the 6th International Symposium on Formal Techniques in Real-Time Fault Tolerant Systems, LNCS 1926, Springer-Verlag, Pune, India, 2000.

[44] F. Tartanoglu, N. Levy, V. Issarny, A. Romanovsky, Using the b method for the formalization of coordinated atomic actions, Tech. Rep. CS-TR: 865, School of Computing Science, University of Newcastle (October 2004).

[45] L. Lamport, Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers, Pearson Education, 2002.

[46] D. Schwier, F. von Henke, R. Stroud, J. Xu, A. Romanovsky, B. Randell, Formalisation of the ca action concept based on temporal logic, in: DeVa - Design for Validation, 2nd year, ESPRIT LTR 20072, 1997, pp. 3–15.

[47] C. Bertolini, L. Brenner, P. Fernandes, A. Sales, A. F. Zorzo, Structured stochastic modeling of fault-tolerant systems., in: Proc. of the 12th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, Vollendam, The Netherlands, 2004, pp. 139–146.

[48] J. Xu, B. Randell, A. B. Romanovsky, R. J. Stroud, A. F. Zorzo, E. Canver, F. W. von Henke, Rigorous development of a safety-critical system based on coordinated atomic actions., in: 29th International Symposium on Fault-Tolerant Computing, 1999, pp. 68–75.

[49] Carnegie Mellon University, The SMV system, http://www.cs.cmu.edu/ modelcheck/smv.html (1998).

[50] A. Capozucca, N. Guelfi, P. Pelliccione, A. Romanovsky, A. F. Zorzo, CAA-DRIP: a framework for implementing coordinated atomic actions., in: Proc. of IEEE International Symposium on Software Reliability Engineering, Raleigh, USA, 2006, pp. 385–394.

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

[52] F. Castor Filho, P. H. da S. Brito, C. M. F. Rubira, Specification of exception flow in software architectures, Journal of Systems and Software 79 (10) (2006) 1397–1418.

[53] P. Krüchten, The 4+1 view model of software architecture, IEEE Software (1995) 42–50.

[54] M. Shaw, P. Clements, A field guide to boxology: Preliminary classification of architectural styles for software systems, in: Proc. of COMPSAC'96, Washington, DC, USA, 1996.

[55] F. Castor Filho, A. Romanovsky, C. M. F. Rubira, Verification of coordinated exception handling, in: Proc. of the 21st ACM Symposium on Applied Computing, Dijon, France, 2006, pp. 680–685.