iArch-U/MC: An Uncertainty-Aware Model Checker for Embracing Known Unknowns

Naoyasu Ubayashi, Yasutaka Kamei and Ryosuke Sato
Kyushu University, Fukuoka, Japan

Keywords: Model Checking, Uncertainty, Known Unknowns, Modular Uncertainty Representation, Partial Model, State Explosion Problem.

Abstract: Embracing uncertainty in software development is one of the crucial research topics in software engineering. In most projects, we have to deal with uncertain concerns by using informal ways such as documents, mailing lists, or issue tracking systems. This task is tedious and error-prone. Especially, uncertainty in programming is one of the challenging issues to be tackled, because it is difficult to verify the correctness of a program when there are uncertain user requirements, unfixed design choices, and alternative algorithms. This paper proposes *iArch-U/MC*, an uncertainty-aware model checker for verifying whether or not some important properties are guaranteed even if *Known Unknowns* remain in a program. Our tool is based on LTSA (Labelled Transition System Analyzer) and is implemented as an Eclipse plug-in.

1 INTRODUCTION

Embracing uncertainty in software development is one of the crucial research topics in software engineering. Garlan, D. discusses the future of software engineering from the viewpoint of uncertainty (Garlan, 2010). He argues that we must embrace uncertainty within the engineering discipline of software engineering. There are two kinds of uncertainty: *Known Unknowns* and *Unknown Unknowns* (Elbaum and Rosenblum, 2014). In *Known Unknowns*, there are uncertain issues in the process of software development. However, these issues are known and shared among the stakeholders including developers and customers. For example, there are alternative requirements although it is uncertain which alternative should be selected. On the other hand, in *Unknown Unknowns*, it is uncertain what is uncertain. It is difficult to deal with *Unknown Unknowns*, because we cannot predict the appearance of this kind of uncertainty. In this paper, we focus on *Known Unknowns* as the first research step.

It is important to systematically describe and verify a program if we have to embrace uncertainties in the software development. Unfortunately, programming under uncertainty is not yet appropriately supported. It would be preferable to be able to modularize uncertain concerns and check whether the important properties concerning to the requirements and designs are satisfied even if there are uncertain concerns. We can continue the development if the properties hold, because the decision can be deferred.

To deal with this problem, we propose the design and implementation of *iArch-U/MC*[1], a model checker for verifying whether or not some important properties such as functionality and deadlock freedom are guaranteed even if *Known Unknowns* remain in a program. Using this uncertainty-aware model checker, we can postpone the decision for dealing with uncertainty to the later software development phase if the selection of uncertain alternatives does not affect the correctness of the properties.

In this paper, we introduce a new programming style to realize an uncertainty-aware model checking: 1) an interface mechanism for modular uncertainty representation based on partial model (Famelis et al., 2012), a single model containing all possible alternative designs of a system; 2) type checker for guaranteeing the refinement simulation between an interface and its source code; and 3) model checker for verifying whether or not a partial model generated from an uncertainty-aware interface satisfies an important

[1]The *iArch-U* (Watanabe et al., 2017) is an IDE (Integrated Development Environment) for supporting uncertainty-aware software development. The *iArch-U/MC*, one of the *iArch-U* tool components, supports model checking. This paper focuses on only the design and implementation of our uncertainty-aware model checker.

Ubayashi, N., Kamei, Y. and Sato, R.
iArch-U/MC: An Uncertainty-Aware Model Checker for Embracing Known Unknowns.
DOI: 10.5220/0006889501760184
In *Proceedings of the 13th International Conference on Software Technologies (ICSOFT 2018)*, pages 176-184
ISBN: 978-989-758-320-9

property. Although model checking is performed at the interface-level (an abstraction of the code), the behavior of a program is also guaranteed in terms of interface descriptions. The reason is that there is a simulation relation between the interface and the code if type checking is passed. Our approach can relax the state explosion problem because of the combinational usage of interface-level model checking and type checking. Our tool is implemented as an Eclipse plug-in and supports uncertainty in Java programming.

This paper is structured as follows. We survey the related work on uncertainty in Section 2. Section 3 introduces an uncertainty-aware interface mechanism. Section 4 shows our approach to type checking and model checking. Section 5 shows the overview of *iArch-U/MC*. We discuss on the originality and the applicability of our approach in Section 6. Concluding remarks are provided in Section 7.

2 RELATED WORK

Recently, uncertainty has attracted a growing interest among researchers. Most of the state-of-the-art studies focus on *Known Unknowns*. As a representative work, a method for expressing *Known Unknowns* using partial model is proposed in (Famelis et al., 2012; Famelis et al., 2015). A partial model is a single model containing all possible alternative designs of a system and is encoded in propositional logic. We can check whether or not a model including uncertainty satisfies some interesting properties. The idea of partial model fits the needs in real software development projects, because alternatives appearing in a software design model or a source program can be represented as a single model by using partial model. This partial model is effective for a developer to manage uncertainty in design and coding phases. For this reason, our approach is based on partial model.

Perez-Palacin, D. and Mirandola, R. provide a systematic review on uncertainty (Perez-Palacin and Mirandola, 2014) and summarize as follows: *The most used definitions of uncertainty simply distinguish between natural variability of physical processes (i.e., aleatory or stochastic uncertainty) and the uncertainties in knowledge of these processes (i.e., epistemic or state-of-knowledge uncertainty).*

The state-of-the-art research themes spread over uncertainty of requirements modeling, software architecture, model transformations, programming, testing, verification, and performance engineering. In (Salay et al., 2013a), partial model is applied to uncertainty in requirements to address the problem of specifying uncertainty within a requirements model, refining a model as uncertainty reduces, providing meaning to traceability relations between models containing uncertainty, and propagating uncertainty-reducing changes between related models. In (Autili et al., 2012; Esfahani et al., 2012; Esfahani et al., 2013; Lago and Vliet, 2005), uncertainty is explored in terms of software architecture. Letier, E et al. present a support method for evaluating uncertainty, its impact on risk, and the value of reducing uncertainty in requirements and architecture (Letier et al., 2014). In (Salay et al., 2013b), a method for change propagation in the context of model uncertainty is proposed. Most of these studies focus on epistemic uncertainty. Uncertain$< T >$, a simple probabilistic programming language for letting programmers without statistics expertise easily and correctly compute with estimates (Bornholt et al., 2014). Uncertain$< T >$ deals with aleatory uncertainty. Elbaum, S. and Rosenblum, D. S. explore how uncertainty affects software testing (Elbaum and Rosenblum, 2014). Uncertainty in self-adaptive systems is explored in (Cheng and Garlan, 2007; Esfahani et al., 2011; Esfahani and Malek, 2013; Perez-Palacin and Mirandola, 2014; Whittle et al., 2010; Yang et al., 2014). Performance and reliability analysis under uncertainty is explored in (Devaraj et al., 2010; K. and S., 2003; Meedeniya et al., 2011; Trubiani et al., 2013).

Uncertainty has been well studied in the field of formal methods: PRISM (Hinton et al., 2006), a probabilistic symbol model checker, can deal with aleatory uncertainty; and three-valued logic consisting of `True`, `False`, and `Undefined` can represent epistemic uncertainty as in VDM (Vienna Development Method) (Fitzgerald and Larsen, 1998). Unfortunately, it is not easy to check whether or not a program behaves correctly at the source code level when there are *Known Unknowns*, because there are many behavioral possibilities in the program.

Although uncertainty is an important research issue, uncertainty in programming and modular reasoning has not been well explored. One of the reasons why uncertainty cannot be dealt with in current programming languages is that the state-of-the-art module mechanisms do not regard an uncertain concern as a first-class pluggable software module. If uncertainty can be dealt with modularly, we can add or delete uncertain concerns to/from code whenever these concerns appear or disappear. Moreover, we can verify the correctness of a program modularly and efficiently if uncertainty is represented modularly. In this paper, we show an uncertainty-aware model checking approach for modular reasoning.

```
[List 1]
01: interface component cPrinter {
02:   public void get();
03:   public void put();
04:   public void print();
05:  [public void utility();]
06: }
07:
08: interface component cScanner {
09:   public void get();
10:   public void put();
11:   public void scan();
12:  [public void utility();]
13: }
14:
15: interface component cCopyMachine {
16:   public void copy();
17: }
```

```
18: interface connector cSystem (
19:   cCopyMachine P, cCopyMachine Q,
20:   cPrinter printer, cScanner scanner) {
21:
22:   GET  = (printer.get -> scanner.get);
23:   PUT  = (printer.put -> scanner.put);
24:   COPY = (scanner.scan -> printer.print);
25:
26:   P.copy = (GET -> COPY -> PUT -> P.copy);
27:   Q.copy = (GET -> COPY -> PUT -> Q.copy);
28: }
```

```
[List 2]
01: interface connector uSystem
02:   extends cSystem (
03:   cPrinter printer, cScanner scanner) {
04:
05:   GET = ({printer.get -> scanner.get,
06:           scanner.get -> printer.get});
07: }
```

Figure 1: Archface-U Description (Printer-Scanner System).

3 MODULAR PROGRAMMING FOR UNCERTAINTY

We adopt an interface mechanism to realize modular programming for uncertainty. In this section, we introduce the interface mechanism called *Archface-U* to represent uncertainty based on partial model by referring our preliminary work (Fukamachi et al., 2015a; Fukamachi et al., 2015b).

Archface-U, an abbreviation of *architectural interface for uncertainty*, represents an abstract program structure in terms of *component-and-connector* architecture consisting of two kinds of interface: *component* and *connector*. Figure 1 (Printer-Scanner System), a well-known parallel system that falls into a deadlock (Magee and Kramer, 2006), is an example of *Archface-U* descriptions. Two processes P and Q acquire the lock from each of the shared resources, the printer and the scanner, and then releases the locks. The symbols $\{\}$ and $[]$ represent *alternative* and *optional*, respectively. A component is the same with ordinary Java interface. A connector, which is specified using the notation similar to FSP (Finite State Processes) (Magee and Kramer, 2006), defines the message interactions among components. In *Archface-U*, uncertain concerns are defined as a sub interface as shown in List 2. By extending the existing interface, we can introduce uncertainty modularly. The `uSystem` interface, an extension of the `cSystem` interface, introduces uncertainty by overwriting the existing `GET` message sequence. In List 2, it is uncertain how to acquire printer and scanner resources in two processes, P and Q. In the overwritten `GET` message sequence, there can be two alternatives: *printer.get* $\rightarrow$ *scanner.get* and *scanner.get* $\rightarrow$ *printer.get*. As shown here, uncertainty can be introduced modularly without invading existing interfaces.

Table 1: Checking Property on Partial Model (Famelis et al., 2012).

$\Phi_M \wedge \Phi_p$	$\Phi_M \wedge \neg\Phi_p$	Property p
SAT	SAT	Maybe
SAT	UNSAT	True
UNSAT	SAT	False
UNSAT	UNSAT	(error)

As shown in Figure 1, we can explicitly represent *Known Unknowns*-type uncertainty using *alternative* and *optional* language constructs. If a developer is writing a program (currently, the target programming language is Java) and he or she becomes aware of the existence of uncertainty, the developer only has to modify *Archface-U* as shown in List 2. The developer does not have to modify the original code, because the essential information containing uncertain concerns is expressed in the *Archface-U* and the behavioral properties can be checked using only this information as explained in Section 4. If an uncertain concern is fixed to certain, a developer only has to delete the corresponding inheritance (List 2) and modify the original *Archface-U* (List 1) if needed. As explained here, uncertainty can be managed at only interface level and the original program code is not basically affected. This is why we adopt an interface mechanism to represent uncertainty modularly.

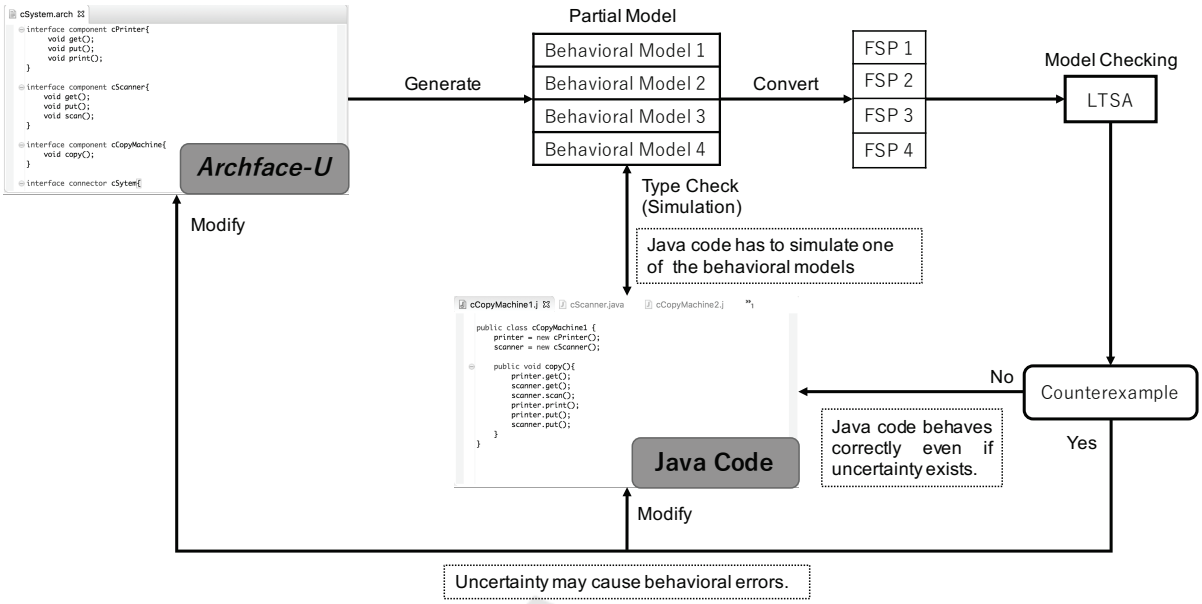

Figure 2: Modular Reasoning for Known Unknowns.

4 MODULAR REASONING BASED ON PARTIAL MODEL

Without modular reasoning about uncertainty, a developer has to rely on global reasoning to check whether some properties are satisfied. In this section, we show the *Archface-U*-based verification in details.

4.1 Uncertainty-aware Verification

We can use the verification power provided by partial model. The behavioral correctness of a program is guaranteed modularly using our compiler (type checker) and model checker. Figure 2 illustrates the verification process. The type checker based on the refinement calculus focusing on simulation checks the conformance between *Archface-U* and its code. The model checker verifies the behavioral properties such as a deadlock by only using the information described in *Archface-U*. Integrating type checker and model checker, we can verify behavioral properties at the code level. Φ_M and Φ_p in Table 1 (Famelis et al., 2012) correspond to logical formula expressing a partial model generated from *Archface-U* and the properties to be checked.

In this paper, we provide two types of true-false decisions for a property p: 1) verified by type checking; and 2) verified by model checking. Φ_p corresponds to the consistency among code or user-defined properties. When a property p is `True`, we can continue to develop even if uncertainty exists. When a property p is `Maybe`, we have to take care of the corresponding properties as a development risk. In other cases, we have to reconsider the code.

4.2 Type Check

Uncertainty is a target of compilation. Our type checker verifies 1) whether a partial model Φ_M generated from *Archface-U* satisfies a property Φ_p such as consistency; and 2) whether code is a subset of the partial model Φ_M (or whether code simulates one of the behavioral models contained in the partial model). It is important that 1) is performed by only *Archface-U* definitions. If code conforms to *Archface-U* in terms of 2), Φ_p is also satisfied in the code. That is, the verification of Φ_p results in modular interface checking. All of the code files are needed for property checking without an interface mechanism provided by *Archface-U*. Fixing the inter-model/code inconsistency is an important problem (Egyed et al., 2008). Our approach can verify inconsistency among code files by type checking even if uncertainty exists. For example, our compiler generates an error message if a method is defined in a component interface and its call is not appeared in the connector interface.

In our compiler, *Archface-U* is translated into a partial model as shown in Figure 2. The followings is the algorithm for *Archface-U* containing *Alternative* uncertainty.

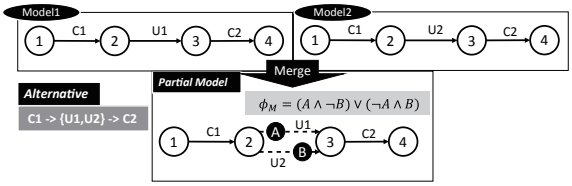

Figure 3: Partial Model Generation (Alternative)

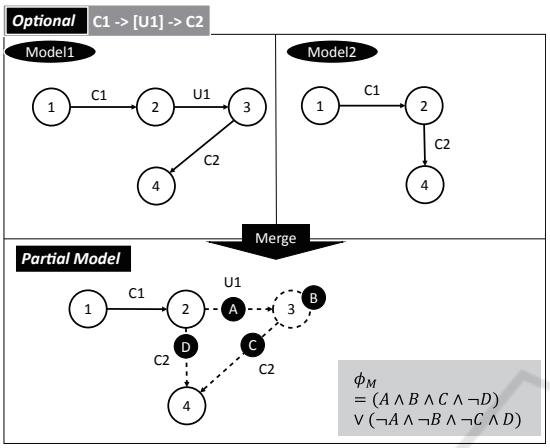

Figure 4: Partial Model Generation (Optional).

STEP 1. Divide a connector interface including *Alternative* to a set of connector interfaces represented by original *Archface-U* that does not contain uncertainty. Each *Archface-U* description represented by LTS (Labelled Transition System) is translated into a state transition model. The upper part of Figure 3 shows the result of transformation in case of $C1 \rightarrow \{U1, U2\} \rightarrow C2$. The number in Figure 3 indicates a state.

STEP 2. State transition models generated in STEP 1 are merged into a state transition machine as shown in the lower part of Figure 3. This state transition model is a partial model. Mandatory edges and nodes appeared in all state transition models are represented by solid lines. Other non-common edges and nodes are represented by dashed lines. After generating a partial model, it is translated into logical formula.

The algorithm in case of *Optional* uncertainty is basically the same to the above algorithm as illustrated in Figure 4. In case of $C1 \rightarrow [U1] \rightarrow C2$, this *Optional* uncertainty is translated into two state transition models as shown in the upper part of Figure 4. This procedure corresponds to STEP 1 in *Alternative* uncertainty. After that, these two models are merged into a state transition machine as shown in the lower part of Figure 4. This procedure corresponds to STEP 2 in *Alternative* uncertainty. $C2$ is represented by two dashed lines, because the source of transition C (state number 3) is different from that of transition D (state number 2).

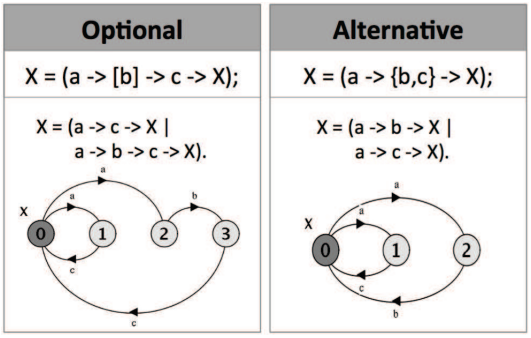

Figure 5: Expansion of uncertain FSP.

4.3 Model Checking Embracing Uncertainty

Behavioral properties represented by LTL can be automatically verified using existing model checkers. In our uncertainty-aware model checker, LTSA (LTS Analyzer) [2] is used as a model checking engine because *Archface-U* is based on FSP supported by LTSA. *Optional* and *Alternative* are translated into ordinary FSP descriptions as shown in Figure 5.

If a property is verified by LTSA and the type check is successfully passed, the program satisfies the property too. Although we used LTSA, our approach takes a standard approach and can be implemented with other popular off-the-shelf checkers such as FDR (Failures Divergences Refinement) [3], a refinement checker for the process algebra CSP (Communicating Sequential Processes).

4.4 Usage Scenario

We explain our verification process using a printer-scanner system as an example. There are four possible resource acquisition sequences as shown in Figure 6. These cases are generated from a partial model described as *Archface-U* (List 2). Type check is passed if the code simulates one of these sequences. The Java code below (List 3) simulates the sequence 1 in Figure 6 and the type check is passed.

[2]http://www.doc.ic.ac.uk/ltsa/, Last accessed 19 April 2018.

[3]https://www.cs.ox.ac.uk/projects/fdr/, Last accessed 19 April 2018.

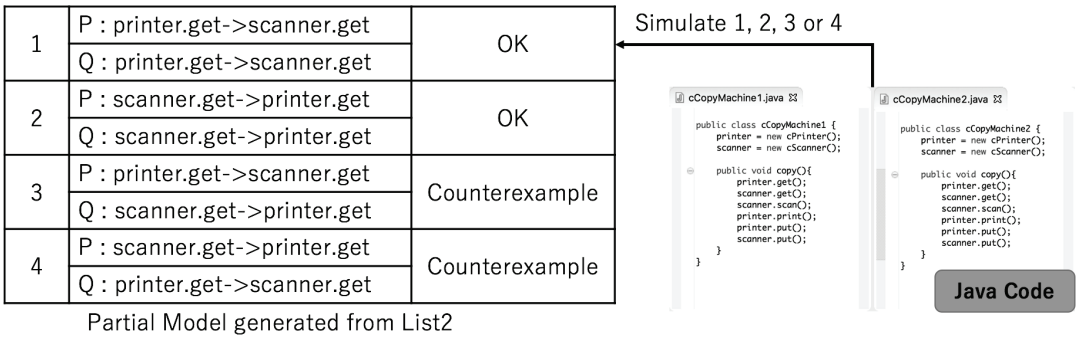

Figure 6: Partial Model and Java Program.

```
[List 3]
 01: public class Printer implements uPrinter {
 02:   public void get()     { ... };
 03:   public void put()     { ... };
 04:   public void print()   { ... };
 05: }
 06:
 07: public class Scanner implements uScanner {
 08:   public void get()     { ... };
 09:   public void put()     { ... };
 10:   public void scan()    { ... };
 11: }
 12:
 13: public class CopyMachine
 14:                 implements uCopyMachine {
 15:   public void copy() {
 16:     printer.get();   scanner.get();
 17:     scanner.scan();  printer.print();
 18:     methodX();
 19:     printer.put();   scanner.put();
 20:   }
 21: }
```

As shown in Figure 2, each behavioral model of a partial model is converted into the corresponding FSP description. Behavioral properties represented by LTL can be automatically verified using our model checker. If counterexamples are not generated by our model checker, we can select any sequence (either of 1, 2, 3, or 4 is OK). We can proceed development even if uncertain concerns exist, because the code simulating any sequence is correct. Of course, List 3 simulating the sequence 1 is correct. Unfortunately, counterexamples are generated in case of the sequences 3 or 4 and these counterexamples show that the acquisition order must be the same. We are notified that uncertainty specified in *Archface-U* (List 2) may cause a deadlock although the code (List 3) is correct. We cannot embrace uncertainty in this scenario. We should not modify List 3 but change List 2 to remove the alternatives of `get` operation orders. After that, We have to run the model checker again and confirm that no counterexamples are generated. As explained here, we can resolve uncertain concerns and make a correct program before debugging and testing.

Our type checker consists of a partial model generator, a refinement verifier, and a consistency verifier. The partial model generator creates a partial model, a set of possible behavior models from FSPs containing *alternative* and *optional* descriptions extended by *Archface-U*. The refinement verifier checks whether the code simulates one model included in the generated partial model. In List 3, a sequence *printer.get* $\rightarrow$ *scanner.get* $\rightarrow$ *scanner.scan* $\rightarrow$ *printer.print* $\rightarrow$ *methodX* $\rightarrow$ *printer.put* $\rightarrow$ *scanner.put* simulates the sequence 1 (Figure 6) generated from the *Archface-U* definitions (List 1 and 2). The call of `methodX` does not violates an LTS defined by FSP in Lists 1 and 2. As a result, properties satisfied by the LTS are also held in the code that passes compile check. The consistency verifier checks the inconsistency not only among *Archface-U* definitions but also among code files. An error is generated if a method is defined in a component interface and its call does not appear in the connector interface. Our approach can verify the inconsistency even if uncertainty exists.

Our compiler adds only type checking embracing uncertainty to the original Java compiler. Compiled code is executable, because *Archface-U* is just a constraint to the code. Program behavior is also guaranteed, because the code simulates just one of the possible models described in *Archface-U*.

4.5 State Explosion Problem

State explosion is a crucial problem when applying model checking to a real project. Especially, it is difficult to apply model checking to source code even if several tools such as CBMC (Bounded Model Checker for C and C++) [4] and Java Pathfinder [5] are already provided. On the other hand, in our approach, model

[4]http://www.cprover.org/cbmc/, Last accessed 19 April 2018.

[5]https://github.com/javapathfinder/, Last accessed 19 April 2018.

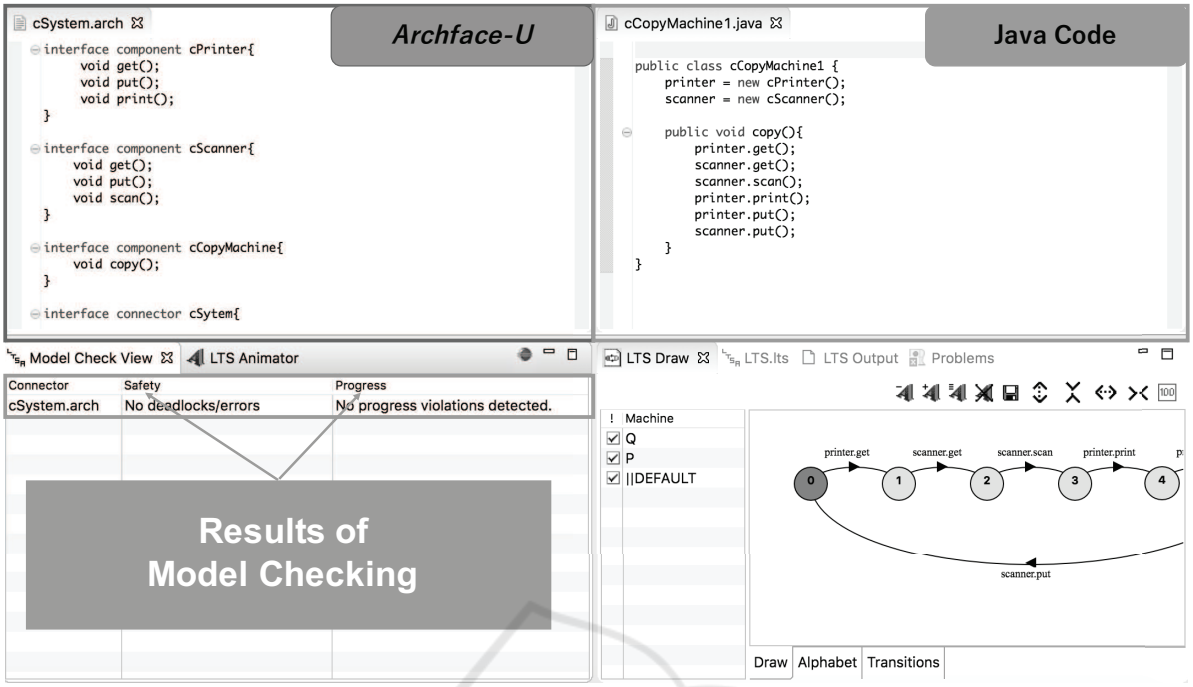

Figure 7: *iArch-U/MC*.

checking is performed in terms of only FSP descriptions in *Archface-U*. Code is not the direct target of model checking. As a result, the number of states is reduced. Nevertheless, as repeatedly claimed, code can be indirectly verified by the model checker if the code conforms to its *Archface-U* via type checker. Our approach mitigates the problem of state explosion by integrating type checking with model checking.

5 TOOL IMPLEMENTATION

The *iArch-U*, an IDE for supporting uncertainty-aware programming, consists of Java program editor, uncertainty-aware compiler, uncertainty-aware model checker, unit testing support, and Git-based uncertainty management support. The *iArch-U* IDE is open source software and its preliminary version can be downloaded from GitHub [6].

The *iArch-U/MC*, a new tool component for model checking, is implemented as an Eclipse plug-in as shown in Figure 7. Figure 2 illustrates the overview of our approach. As roughly explained in Section 4, a partial model is generated from *Archface-U* definitions. Each behavioral model consisting the partial model is converted into the corresponding FSP description. Behavioral properties represented by LTL can be automatically verified using LTSA supporting FSP. If a property is verified by a model checker and the type check is successfully passed, the program satisfies important properties such as deadlock freedom. When a property is `True`, we can continue to develop even if uncertainty exists. Otherwise, we have to reconsider the code. Using *iArch-U/MC*, we can explore which uncertainty can be permitted by interactively modifying not program code itself but *Archface-U* descriptions and checking the behavioral correctness.

6 DISCUSSION

In this section, we discuss on the originality and the applicability of our idea.

Someone might consider that *Archface-U* is similar to variability modeling in SPL (Bąk et al., 2014). How different is our approach from SPL? If there is no difference, we can deal with uncertainty by only using SPL technologies. Indeed, uncertainty in structural aspects (a component interface in *Archface-U*) can be represented by defining uncertain features in a feature model. Although it is difficult to represent behavioral aspects of uncertainty (a connector interface in *Archface-U*) in a feature model, there are studies on behavioral variability (Classen et al., 2012; Ghezzi and Sharifloo, 2011).

Our most important contribution is to introduce

[6] http://posl.github.io/iArch/, Last accessed 19 April 2018.

the *interface-based variability* to the world of SPL. As claimed in this paper, this interface enables the valuable integration of code-conformance check via type checker and model checking taking into account abstraction. Only using current SPL technologies, it is not easy to integrate important facilities mentioned above. Moreover, our idea can be basically applied to SPL by not limiting to variability in uncertain concerns. Similarity to SPL comes from the characteristics of *Known Unknowns* in which uncertainty is a subset of variability.

Although the interface mechanism of *Archface-U* can be applied to SPL, the process of SPL is different from that of uncertainty-aware software development. The former focuses on generating a product from a set of features represented by *optional* and *alternatives*. Product structure does not basically change through a software development, although product regeneration may occur several times to deal with small changes. On the other hand, our approach focuses on the management of uncertainty frequently appearing or disappearing in a software development. Product generation is out of scope. The main concern is to verify whether some important properties are guaranteed even if uncertainty exists and to decide whether resolution of uncertainty can be postponed. For this reason, modular reasoning realized by type checking and model checking is important.

When modifying a large product, it is necessary to impose constraints among alternatives because a large number of checking for relatively complex systems would result in errors. It makes sense to extend the *alternative* and *optional* operators to include constraints. However, the number of constraints might be overwhelming due to combinatorial explosion. The lack of precision may indeed identify more constraints than necessary in order to keep a sound verification. This issue is our future work.

7 CONCLUSIONS

In this paper, we proposed *iArch-U/MC*, an uncertainty-aware model checker for verifying whether or not some important properties are guaranteed even if *Known Unknowns* remain in a program. Our approach deals with epistemic uncertainty at the program code level. As the next step, we plan to integrate *iArch-U/MC* with LTSA-PCA (Probabilistic Component Automata) (Rodrigues et al., 2014) to support aleatory uncertainty.

ACKNOWLEDGMENTS

We thank Syunya Nakamura, Keisuke Watanabe, and Takuya Fukamachi for their great contributions. They were students of Naoyasu Ubayashi. This work was supported by JSPS KAKENHI Grant Numbers JP26240007.

REFERENCES

Autili, M., Cortellessa, V., Ruscio, D. D., Inverardi, P., Pelliccione, P., and Tivoli, M. (2012). Integration architecture synthesis for taming uncertainty in the digital space. In *Proceedings of the 17th Monterey Conference on Large-Scale Complex IT Systems: Development, Operation and Management, pp.118-131.*

Bąk, K., Diskin, Z., Antkiewicz, M., Czarnecki, K., and Wąsowski, A. (2014). Clafer: Unifying class and feature modeling. In *Software & Systems Modeling, December 2014, pp.1-35.*

Bornholt, J., Mytkowicz, T., and McKinley, K. S. (2014). Uncertain$< t >$: A first-order type for uncertain data. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2014), pp.51-66.*

Cheng, S. W. and Garlan, D. (2007). Handling uncertainty in autonomic systems. In *Proceedings of the International Workshop on Living with Uncertainties (IWLU 2007).*

Classen, A., Cordy, M., Heymans, P., Legay, A., and Schobbens, P. Y. (2012). Model checking software product lines with snip. In *International Journal on Software Tools for Technology Transfer, 14(5), pp.589-612.*

Devaraj, A., Mishra, K., and Trivedi, K. S. (2010). Uncertainty propagation in analytic availability models. In *Proceedings of the Symposium on Reliable Distributed Systems (SRDS 2010), pp.121-130.*

Egyed, A., Letier, E., and Finkelstein, A. (2008). Generating and evaluating choices for fixing inconsistencies in uml design models. In *Proceedings of the 23rd International Conference on Automated Software Engineering (ASE 2008), pp.99-108.*

Elbaum, S. and Rosenblum, D. S. (2014). Known unknowns: Testing in the presence of uncertainty. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014), pp.833-836.*

Esfahani, N., Kouroshfar, E., and Malek, S. (2011). Taming uncertainty in self-adaptive software. In *Proceedings of the 8th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2011), pp.234-244.*

Esfahani, N. and Malek, S. (2013). Uncertainty in self-adaptive software systems. In *Software Engineering for Self-Adaptive Systems II, volume 7475 of LNCS, pp.214-238.* Springer.

Esfahani, N., Malek, S., and Razavi, K. (2013). Guidearch: Guiding the exploration of architectural solution space under uncertainty. In *Proceedings of the 35th International Conference on Software Engineering (ICSE 2013), pp.43-52*.

Esfahani, N., Razavi, K., and Malek, S. (2012). Dealing with uncertainty in early software architecture. In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering (FSE 2012), pp.21:1-21:4*.

Famelis, M., Ben-David, N., Sandro, A. D., Salay, R., and Chechik, M. (2015). Mu-mmint: an ide for model uncertainty. In *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015), Demonstrations Track, pp.697-700.*

Famelis, M., Salay, R., and Chechik, M. (2012). Partial models: Towards modeling and reasoning with uncertainty. In *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012), pp.573-583*.

Fitzgerald, J. and Larsen, G. P. (1998). *Modeling Systems, Practical Tools and Techniques in Software Development*. Cambridge University Press.

Fukamachi, T., Ubayashi, N., Hosoai, S., and Kamei, Y. (2015a). Conquering uncertainty in java programming. In *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015), Poster Track, pp.823-824*.

Fukamachi, T., Ubayashi, N., Hosoai, S., and Kamei, Y. (2015b). Modularity for uncertainty. In *Proceedings of the 7th International Workshop on Modelling in Software Engineering (MiSE 2015), pp.7-12*.

Garlan, D. (2010). Software engineering in an uncertain world. In *Proceedings of FSE/SDP Workshop on Future of Software Engineering Research (FoSER 2010), pp.125-128*.

Ghezzi, C. and Sharifloo, A. M. (2011). Quantitative verification of non-functional requirements with uncertainty. In *Dependable Computer Systems, pp.47-62*.

Hinton, A., Kwiatkowska, M., Norman, G., and Parker, D. (2006). Prism: A tool for automatic verification of probabilistic systems. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2006), pp.441-444*.

K., G.-P. and S., K. (2003). Assessing uncertainty in reliability of component-based software systems. In *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE 2003), pp.307-320.*

Lago, P. and Vliet, H. (2005). Explicit assumptions enrich architectural models. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005), pp.206-214*.

Letier, E., Stefan, D., and Barr, E. T. (2014). Uncertainty, risk, and information value in software requirements and architecture. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014), pp.883-894*.

Magee, J. and Kramer, J. (2006). *Concurrency: State Models & Java Programs Second Edition*. Wiley.

Meedeniya, I., Moser, I., Aleti, A., and Grunske, L. (2011). Architecture-based reliability evaluation under uncertainty. In *Proceedings of the 7th International ACM Sigsoft Conference on the Quality of Software Architectures (QoSA 2011), pp.85-94*.

Perez-Palacin, D. and Mirandola, R. (2014). Uncertainties in the modeling of self-adaptive systems: a axonomy and an example of availability evaluation. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014), pp.3-14*.

Rodrigues, P., Lupu, E., and Kramer, J. (2014). Ltsa-pca: Tool support for compositional reliability analysis,. In *ICSE Companion 2014 Companion Proceedings of the 36th International Conference on Software Engineering (ICSE 2014), pp.548-551*.

Salay, R., Chechik, M., Horkoff, J., and Sandro, A. D. (2013a). Managing requirements uncertainty with partial models. In *Requirements Engineering, Volume 18, Issue 2, pp.107-128*.

Salay, R., Gorzny, J., and Chechik, M. (2013b). Change propagation due to uncertainty change. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE 2013), pp.21-36*.

Trubiani, C., Meedeniya, I., Cortellessa, V., Aleti, A., and Grunske, L. (2013). Model-based performance analysis of software architectures under uncertainty. In *Proceedings of the 9th International ACM Sigsoft Conference on the Quality of Software Architectures (QoSA 2013), pp.69-78*.

Watanabe, K., Ubayashi, N., Fukamachi, T., Nakamura, S., Muraoka, H., and Kamei, Y. (2017). iarch-u: Interface-centric integrated uncertainty-aware development environment. In *9th International Workshop on Modelling in Software Engineering (MiSE 2017), pp.40-46.*

Whittle, J., Sawyer, P., Bencomo, N., Cheng, B. H. C., and Bruel, J. M. (2010). Relax: A language to address uncertainty in self-adaptive systems requirement. In *Requirements Engineering, 15(2), pp.177-196*.

Yang, W., Xu, C., Liu, Y., Cao, C., Ma, X., and Lu, J. (2014). Verifying self-adaptive applications suffering uncertainty. In *Proceedings of the 29th International Conference on Automated Software Engineering (ASE 2014), pp.199-210.*